



INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :

G06F 9/46

A1

(11) International Publication Number:

WO 98/02813

(43) International Publication Date:

22 January 1998 (22.01.98)

(21) International Application Number: PCT/US97/11887

(22) International Filing Date: 10 July 1997 (10.07.97)

(30) Priority Data:

08/678,317

11 July 1996 (11.07.96)

US

(71) Applicant: TANDEM COMPUTERS INCORPORATED
[US/US]; 10435 North Tantau Avenue, Loc. 200-16,
Cupertino, CA 95014 (US).(72) Inventors: DE BORST, Jeroen, Peter; Alte Mauergasse 5, D-
61348 Bad Homburg (DE). BONHAM, Peter, Douglas;
Am Alten Bach 19, D-61352 Bad Homburg (DE). ER-
LENKOETTER, Ansgar; Auf der Heide 44, D-61267 Neu-
Anspach (DE). SCHOFIELD, Andrew, Charles; Linden-
bühl 27, CH-6330 Cham (CH). KAESER, Reto, Richard;
Steinhügelstrasse 40, CH-8968 Mutschellen (CH).(74) Agents: GRANATELLI, Lawrence, W. et al.; 600 Hansen Way,
Palo Alto, CA 94304 (US).(81) Designated States: JP, European patent (AT, BE, CH, DE, DK,
ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published

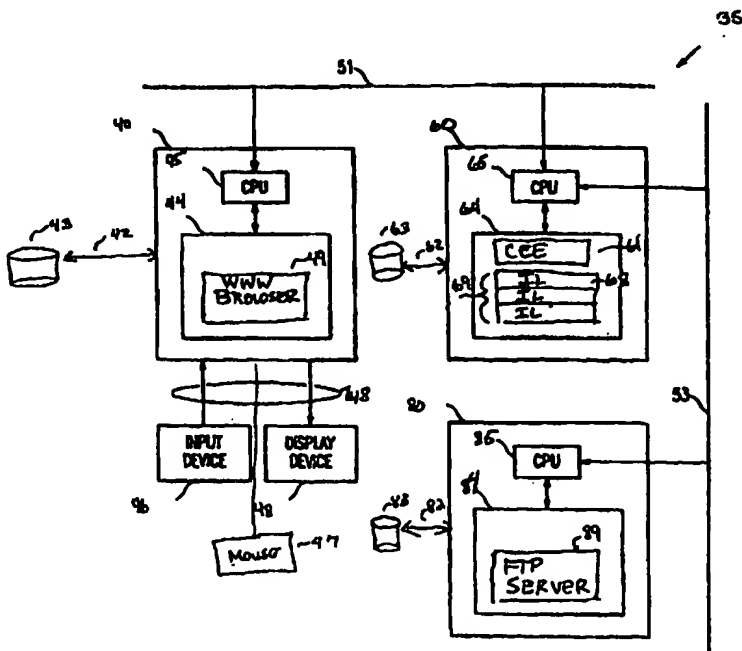
With international search report.

Before the expiration of the time limit for amending the
claims and to be republished in the event of the receipt of
amendments.

(54) Title: OBJECT-ORIENTED METHOD AND APPARATUS FOR INFORMATION DELIVERY

(57) Abstract

An object-oriented method and apparatus for delivering information from one component to another across a network of computers includes the steps of loading implementation libraries for adapter and information provider components into memory and creating factory objects for those components. When a request arrives over the network, the factory objects are called and stream objects are created by the factory objects. Data is then streamed from an information provider source to the original requestor using the stream objects.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakhstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

OBJECT-ORIENTED METHOD AND APPARATUS FOR INFORMATION DELIVERY

BACKGROUND OF THE INVENTION

1. Field of the Invention

5 The present invention relates to an object-oriented method and apparatus for delivering information within a computer or across a network of computers. More particularly, the invention relates to a distributed object system using a factory/stream model for delivering data and related context from one component in the computer or network to another.

10 2. Background

Distributed computing consists of two or more pieces of software sharing information with each other. These two pieces of software could be running on the same computer or on different computers connected to a common network. Most distributed computing is based on a client/server model. With the client server model, two major types
15 of software are utilized: client software, which requests the information or service, and server software, which provides the information or service.

Information or services are usually delivered from server to client by Information Delivery System software applications. Such applications are often monolithic, protocol-specific, UNIX-based server applications consisting of a module that awaits a request for
20 information from a client application. Once that request arrives, the module will copy (or "replicate") itself and the copy of the module will process the request. The request will be processed by the copy, for example, by providing information from an SQL database or information contained in a local disk file. Meanwhile, the original module will continue to monitor for incoming requests.

25 This information delivery system architecture, however, is usually protocol-specific. In other words, the delivery system application only supports a particular protocol, such as HTTP, TCP/IP, or SPX. Thus, requests arriving via an unsupported protocol cannot be serviced. Such systems are inherently inflexible and non-extensible.

Attempts by monolithic applications to support multiple protocols further

demonstrate their non-extensibility. Generally, however, adding support for several protocols requires adding additional code in the application. As more and more protocols are created, the application becomes correspondingly larger. If the application is executing a single process, the application may run without errors. If, however, the application is running multiple processes, too many system resources are utilized and the application crashes.

Moreover, the lack of fault tolerance and error-handling in many of these systems makes correction almost impossible. When system resources are depleted and the application terminates, it is often difficult to determine which client request triggered the breakdown. For instance, the system has no built-in mechanism for determining whether an SQL query or a World Wide Web page request caused the error. Accordingly, the application cannot be readily debugged and corrected.

One possible solution is the use of a distributed object system. Distributed object computing combines the concepts of distributed computing (described above) and object-oriented computing. Object-oriented computing is based upon the object model where pieces of code called "objects"—abstracted from real objects in the real world—own attributes and provide services through methods (or "operations" or "member functions"). Typically, the methods operate on the private attributes (data) that the object owns. A collection of like objects make up an interface (or "class" in C++ parlance). Each object is identified by a unique identifier called an object reference.

In a distributed object system, a client sends a request (or performs an "object call") containing an indication of the operation for the server to perform, the object reference, and a mechanism to return "exception information, about the success or failure of a request. In addition, certain "context" information concerning the client (such as the platform or machine) may be included in the request. The server receives the request and, if possible, carries out the request on the specific object and returns information regarding the success or failure of the operation ("exception information"). Both client and server must have information about the available objects and operations that can be performed. Accordingly, both must have access to a common language, such as the Interface Definition Language (IDL), as defined by the Object Management Group (OMG). Interface definitions are usually written in IDL, compiled, and linked into the client and server applications.

Distributed object computing solves some of the problems associated with the prior

art monolithic applications. With distributed objects, large applications can be broken down into smaller "components", making the system more manageable and less subject to breakdown. Moreover, the system allows various components to plug-and-play, interoperate across networks, run on different platforms, coexist with legacy applications through object wrappers, roam on networks, and manage themselves and the resources they control. In addition, errors may be caught using exception handling.

Unfortunately, the current standard architectures for distributed systems have not specifically addressed how information can be provided from numerous information providers to clients over multiple protocols. The Common Object Request Broker Architecture (CORBA) proposed by OMG utilizes an object request broker (ORB) to handle object calls between a client and a server. The CORBA standard, however, remains tied to the use of a specific protocol. Other standards, such as OpenDoc and OLE, are similarly protocol-specific.

Furthermore, in traditional distributed object systems, a classical object call is based upon a request/response mechanism. Accordingly, only certain amounts of information may be transmitted in a particular object call. Prior art information delivery systems offer no mechanism for supporting delivery of large amounts of data (4 GB of video data, e.g.).

Accordingly, a need exists for a method for delivering information that supports multiple protocols.

Further, a need exists for a method for delivering information that uses distributed object technology to promote the use of individual components.

Further, a need exists for a method for delivering large amounts of data across a network.

SUMMARY OF THE INVENTION

The present invention is directed to an object-oriented information method and system having support for multiple protocols. The information delivery system of the present invention further permits the delivery of large amounts of information to client applications across a network.

In a preferred embodiment, the object-oriented method for delivering information includes the following steps. First, adapter components for requesting information, information provider components, and navigator components are loaded into the memory of

one or multiple computers. A navigator object, adapter factory object, and information provider factory object are created. When a request arrives from a requestor on the network, the factory objects are called to create stream objects. The stream objects are used to stream information from an information source to the requestor. The stream interface
5 includes operations for writing and reading data to and from an information source. By utilizing this model, data can be streamed in discrete portions from the information source to the requestor.

In another embodiment, an object-oriented information delivery system of the present invention includes at least one information provider component that provides
10 information via a server or a gateway to another server and at least one adapter component for requesting information from the information provider component via a request. Each component is implemented as an implementation library. Numerous adapter components can be utilized in the system corresponding to the various protocols that are available. Each adapter component is equipped with numerous sub-components, including a dispatcher
15 component that dispatches the request to the other sub-components and a navigator component that decides what information provider to utilize to fulfill the request. Finally, the system utilizes object-oriented technology for discretely sending packets of information from the information provider component to the adapter component. In particular, the system uses a factory/stream model where each adapter and information provider component
20 implements a factory interface that allows stream objects to be created and a stream interface that allows large units of information to be retrieved or stored.

The information delivery system can also utilize a trader component that acts as an intermediary between the adapter components and the information provider component. If the trader is used, the navigator sub-component selects the trader and the trader selects the
25 correct information provider. The trader component can also be replicated and the navigator sub-component then selects the proper trader.

The information delivery method and apparatus of the present invention facilitates the creation of scalable and extensible components that can be adapted to numerous information delivery scenarios. More significantly, the object-oriented aspect of the present
30 invention allows various components to deliver information in a protocol-independent manner across heterogeneous platforms.

A more complete understanding of the information delivery system will be afforded

to those skilled in the art, as well as a realization of additional advantages and objects thereof, by a consideration of the following detailed description of the preferred embodiment. Reference will be made to the appended sheets of drawings which will first be described briefly.

5

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a block diagram of a network implementing the method of the present invention.

Fig. 2 is a Common Execution Environment capsule.

10 Fig. 3 is a block diagram showing the compilation and linking of IDL source files.

Fig. 4 is a representation of a context data structure used in the method of the present invention.

Fig. 5 is an architectural overview of the Information Matrix.

Fig. 6 is a sample HTTP adapter.

15 Fig. 7 is a sample Gopher gateway.

Fig. 8 is a chart depicting component interaction without trading.

Fig. 9 is a chart depicting component interaction with trading.

Fig. 10a is a chart depicting the start-up and set-up phases for Adapter component interaction.

20 Fig. 10b is a chart depicting the stream and clean-up phases for Adapter component interaction.

Fig. 11 is a chart depicting trader interaction.

Fig. 12 is a chart depicting Information Provider component interaction.

Fig. 13 is a chart depicting transformer component interaction.

25 Figs. 14(a) and 14(b) show an IDL-defined data structure, generated CIN description, and generated array of op_tag structures.

Fig. 15 is a flow chart describing a first preferred embodiment of the client side of the method of the present invention.

30 Fig. 16 is a flow chart describing a first preferred embodiment of the server side of the method of the present invention.

Fig. 17 is a flow chart depicting the generation of a CIN descriptor for base and compound data types.

Fig. 18 is a flow chart depicting the generation of a CIN descriptor for an operation.

Fig. 19 is a flow chart depicting the generation of a CIN descriptor for an interface.

Fig. 20 is a diagram showing a PIF data structure.

Fig. 21 is a diagram showing an entry data structure.

5 Fig. 22 is a diagram showing an operation data structure.

Fig. 23 is a diagram showing a union data structure.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

10 I. Hardware Overview

Figure 1 shows a sample network 35. The network 35 includes a client computer 40, an Information Matrix ("IM") computer 60 using the information delivery method and apparatus of the present invention, and a remote file server computer 80. The client computer 40, IM computer 60, and file server computer 80, are connected by network
15 connections 51, 53, such as internet connections. The client computer 40 communicates over a bus or I/O channel 42 with an associated disk storage subsystem 43 and via an I/O channel 48 with an input device 41, such as a keyboard, display device 48, such as a video display terminal ("VDT"), and mouse 47. The client computer 40 includes a CPU 45 and a
20 memory 44 for storing current state information about program execution. A portion of the memory 44 is dedicated to storing the states and variables associated with each function of the program which is currently executing on the client computer 40. The client computer memory 44 includes a World Wide Web ("WWW") browser 49, such as the browser sold under the trademark Netscape Navigator by Netscape Communications Corp.

The file server computer 80 communicates over a bus or I/O channel 82 with an
25 associated disk storage subsystem 83. The server system 80 includes a CPU 85 and a memory 84. The file server memory 84 includes a file transport protocol ("FTP") server 89 loaded therein. The FTP server 89 provides other computers (such as the IM Computer 60 and the client computer 40) with files from the disk subsystem 83 or from other remote computers that communicate with the file server computer 80.

30 The IM computer 60 communicates over a bus or I/O 62 with an associated disk storage subsystem 63. The IM computer 60 includes a CPU 65 and a memory 64. The memory 64 includes one or more implementation libraries 68 and an execution environment

("CEE") 61 (as discussed below).

The network 35 as shown in Figure 1 is merely demonstrative of a typical network. The method and apparatus of the present invention uses distributed software which may be distributed over one or more computers. Thus, the implementation libraries 68 and execution environment 61 may be stored in the client computer memory 44 or the file server computer memory 84. Furthermore, the World Wide Web browser 49, FTP server 89, and execution environment 61 may be loaded on a single computer.

II. Distributed Computing Environment

The method and apparatus of the present invention may be utilized within any distributed computing environment. In a preferred embodiment, the Common Execution Environment ("CEE") 61, which is a component of the Tandem Message Switching Facility ("MSF") Architecture, is used. The CEE activates and deactivates objects and is used to pass messages between implementation libraries loaded in CEE "capsules". The CEE may be stored in the memory of a single machine, as shown in Figure 1. More likely, however, copies of the CEE and implementation libraries will be loaded on multiple machines across a network.

The CEE uses a "capsule" infrastructure. A capsule encapsulates memory space and one or more execution streams. A capsule may be implemented differently on different systems depending upon the operating system used by the system. For instance, on certain systems, a capsule may be implemented as a process. On other systems, the capsule may be implemented as a thread.

Figure 2 shows a CEE capsule 70 contained in, for example, an IM computer 60 that includes the CEE 61 and certain of the core CEE components and implementations of objects contained within Implementation Libraries 68. The Implementation Libraries 68 include hand-written code, such as a client or server application 79 and client stubs 77 generated from the IDL specification of the object's interface, as described below. The Implementation Libraries 68 and the CEE 61 interact through the down-calling of dynamically-accessible routines supplied by the CEE 61 and the up-calling of routines contained in the Implementation Library. The CEE 61 can also receive object calls 82 from other capsules within the same machine and requests 84 from other CEE's.

Objects implemented in a CEE capsule may be configured or dynamic. Configured

objects have their implementation details stored in a repository (such as the MSF Warehouse 75) or in initialization scripts. Given a request for a specific object reference, the CEE 61 starts the appropriate capsule based on this configuration data. The capsule uses the configuration data to determine which Implementation Library to load and which object initialization routine to call. The object initialization routine then creates the object. Dynamic objects are created and destroyed dynamically within the same capsule. Dynamic objects lack repository-stored or scripted configuration information.

The following paragraphs describe a system-level view of how the Implementation Libraries 68 interact with the CEE 61. The CEE 61 implements requests to activate and deactivate objects within a capsule. In addition, the CEE facilitates inter-capsule object calls 72 as well as requests from other CEE's 74, as discussed above. Object activation requests arise when an object call from a client or server application must be satisfied. To activate an object, the CEE 61 loads the appropriate Implementation Library (if not already loaded) containing the object's operations and then calls a configured object initialization routine contained in the Implementation Libraries 68, as discussed below. The initialization routine specifies which interface the Implementation Libraries support and registers the entry points of the object's operations to be called by the CEE 61 at a later time.

The Implementation Libraries 68 are loaded at start-up. During start-up, the CEE 61 runs its own initialization routine. This initialization routine tells the CEE 61 where to locate the various Implementation Libraries 68. Once located by the CEE 61, the CEE 61 calls the initialization routines in the Implementation Libraries 68. The initialization routines contained in the Implementation Libraries must first carry out any required application-specific initialization (e.g., opening files). Next, both the initialization routines call a generated stub function which, in turn, down-calls a CEE function (contained in a dynamic library as stated above) called CEE_INTERFACE_CREATE to specify the object's interface. An interface may be specified for each object. The interface description is normally generated from an IDL description of the interface, as discussed below. CEE_INTERFACE_CREATE creates an interface and returns an "interface handle" to the newly created interface. The handle is a unique identifier that specifies the interface. The initialization routine then uses the interface handle to down-call CEE_IMPLEMENTATION_CREATE. CEE_IMPLEMENTATION_CREATE returns an "implementation handle," that is a unique identifier specifying the implementation for each

operation in the interface. Finally, the initialization routine uses the implementation handle to call a stub function which down-calls CEE_SET_METHOD. CEE_SET_METHOD specifies the actual addresses of specific method routines of the implementation as contained in the implementation libraries 68.

5

III. Compiling and Linking IDL Source Files

Figure 3 shows how Interface Definition Language ("IDL") source files are compiled into the implementation libraries that are used in the method and apparatus of the present invention. First, an IDL source file 101 is prepared containing IDL interface definitions. An IDL compiler 103 compiles the source file 101. The IDL compiler 103
10 parses the code 101 to produce an intermediate Pickled IDL file ("PIF") file 105 for storage of the original source file. A code generator 111 then parses the PIF file. The creation of a PIF file is described below in Section XXX. Alternatively, the IDL compiler and code generator may be combined to generate code. The code generator 111 generates files in the
15 language of the client and server applications. If the client and server applications are in different languages, different code generators 111 are used. Alternatively, the code generator 111 and the IDL compiler 103 may be combined in a single application to produce language-specific code. The code generator 111 produces a client stub file 77 containing client stub functions and a server stub file 87 containing definitions of object
20 implementations. The client stub file 77 and the server stub file 87 are compiled by programming language-specific compilers 121, 123 to produce compiled client stub object code and compiled server stub object code. Similarly, a client application 79 and a server application 89 are compiled by programming-language-specific compilers to produce compiled client application object code and compiled server application object code. The
25 client application 79 and the server application 89 also include a header file 119 generated by the code generator 111. The header file 119 contains common definitions and declarations. Finally, language compiler 121 links the client application object code and the client stub object code to produce an implementation library 71. Similarly, second language compiler 123 links the server application object code and the server stub object code to
30 produce another implementation library 81.

IV. Factory and Stream Interfaces

The method and apparatus of the present invention are used within an information delivery system ("Information Matrix") composed of one or more software components that can be distributed across a network. Each component can be implemented in a variety of ways. The method and apparatus of the present invention permits a developer to implement each component to suit particular information delivery tasks. The preferred implementation for each component, however, is as part of an Implementation Library 68 that is loaded and executed within the CEE 61. Each implementation library 68 implements IDL-defined interfaces to create objects of those interfaces. Interface descriptions are written in IDL, then compiled by an IDL compiler and linked into the code of the component by a code generator, as discussed above. The component then creates an object of that interface. Each object interacts with another object through object calls.

As shown in Figure 5, the components of the Information Matrix 69 can be divided into four primary abstract components (abstractions): (1) Adapters; (2) Servers; (3); Gateways; and (4) Traders. Each Abstraction includes one or more actual software components (Implementation Libraries). In the examples described herein, all of the abstractions and their included components are loaded on a single IM computer. It is possible, however, for certain components to be loaded on the client computer and for certain components to be loaded on the server computer. The Adapter abstraction provides a method for requestors to retrieve information from the Information Matrix 69. In Figure 1, the World Wide Web browser 49 loaded in the client machine memory would be the requestor. The request would arrive at one of the components of the Adapter abstraction loaded in the IM computer memory 64. Figure 5 shows sample Adapter abstractions, including a Hypertext Transport Protocol ("HTTP") Adapter 131, a Gopher (Internet server) protocol Adapter 133, and a Network News Transport Protocol Adapter 135. Additional Adapter abstractions could be added for other types of information retrieval requests. The Gateway abstraction connects external information sources to the information system. Figure 5, for example, shows an HTTP Gateway 137, a Gopher Gateway 139, and a File Transport Protocol ("FTP") Gateway 141. Servers 143 reside inside the information delivery system to provide information from a local source, such as a disk file 145. Gateways and Servers are referred to as "information providers" since both provide information to Adapters (Gateways provide information from an external source, while

Servers provide information from an internal source). Finally, a Trader 150 selects one instance of a required type of Gateway or Server based upon a method that attempts to balance the load on information providers. Information may be provided to requestors, however, without the use of a Trader 150.

Each component of the present invention can implement numerous IDL interfaces. The present invention requires, however, that, for certain components, two interfaces be implemented: the Factory interface and the Stream interface. For certain components that implement the Factory and Stream interfaces, the interfaces are implemented exactly as defined below. Other components, however, "inherit" the operations of the Factory and Stream interfaces, i.e., the component adds operations to these interfaces. The Factory and Stream interfaces are utilized by certain components to stream large amounts of data from one component to another.

The Factory interface is called by an object (of any component) to create a Stream object that, in turn, is used to deliver large amounts of data between components. The Factory interface contains only a Create operation and raises only one exception (an unexpected occurrence). The Create operation is defined in IDL as follows:

```
void Create ( in String          path,
              in unsigned long   factoryType,
              inout unsigned long contextBuffer,
              out unsigned long   streamType,
              out Stream          streamObject,
              out unsigned long   streamHandle
            )
    raises (Exception);
```

The first parameter, *path*, is the name of the resource for which a stream of information needs to be created. The second parameter is a *factoryType* parameter that contains values indicating that the factory operation needs to find a resource and create a stream for that resource. The *contextBuffer* parameter contains the "Context" for the stream requested and can contain data that may cause the factory to behave differently. For instance, the context may include a request for data of a particular type (.JPEG, .GIF, e.g.). (A preferred configuration for the propagation of context is discussed below). The Create operation has

three output parameters. A *streamType* parameter indicates the direction of the stream for the returned Stream object. The *streamType* parameter can have three possible values indicating whether the stream is for a "Get" operation only, for a "Put" operation only, or whether both operations can be performed. Each of these operations is discussed in detail below. The *streamObject* parameter contains an object reference for the requested Stream object. All subsequent stream operations must be addressed to this object. The *streamHandle* parameter identifies the stream of information within the Stream object. The *streamHandle* is provided to the Stream object to obtain information from the object.

The Factory interface raises only one exception if an error occurs. In the preferred embodiment, an exception is raised for one or more of the following reasons: (1) the stream table is full; (2) the process does not have access to the required information for security reasons; (3) the requested resource does not exist; (4) the requested resource is not available (due to a locked file or a failed remote host, e.g.); (5) the requested resource is not available in one of the requested formats (the file is a .JPEG file and the caller specified that it would only accept .GIF files, e.g.); and (5) the operation was successful but no data is present.

The Stream interface supports the streaming of information from one component to another component. Object calls are limited to a certain size depending upon the underlying mechanism used and its implementation on a specific system. The Stream interface facilitates the retrieval of information significantly larger than normal through multiple calls to an object's "Get" operation, as described below.

A Stream object is called by other objects to obtain discrete units of information. The Stream interface contains four operations: Get, Put, Destroy, and Cancel. The Get operation is defined in IDL as follows:

```

void Get (    inout unsigned long    opID,
             in unsigned long        handle,
             inout unsigned long     offset,
             in unsigned long        length,
             out unsigned long        buffer,
             out boolean              endofStream,
             )
    raises (Exception);

```

The Get operation is used by a component to read data from a Stream object ("stream" or "data stream") identified by a "handle." Data can be read sequentially or randomly. The Get operation takes *handle*, *length*, *buffer*, *end of stream*, *operation* identifier and offset parameters. The *handle* parameter identifies the requested stream of information. The handle for a stream of data is returned from the call to the Create operation on a Factory object. A *length* parameter specifies the amount of data the calling component is willing to accept. The maximum value allowed for the *length* is defined by the transport mechanism used for the object call. The actual amount of data returned may be less depending upon the implementation. The *buffer* parameter contains the requested data upon a successful return from the operation. The *end of stream* parameter (usually a boolean) indicates whether more data exists within the requested stream. The *offset* parameter describes the start of the block of data that is to be read. This value can be a byte position (if the particular implementation supports random data access) or a value that indicates that the data block starts right after the previous data block read from the stream. The *operation* identifier simply identifies the read operation.

The Put operation is called to write data to the stream identified by the handle. Data can be written sequentially or, if the implementation permits, in random order. The Put method is defined as follows:

```

void Put (    inout unsigned long   opID,
            in unsigned long        handle,
            in unsigned long        offset,
            in unsigned long        buffer
            )
    raises (Exception);

```

As the definition shows, the Put operation takes four parameters. The parameters are similar to their counterparts in the Get method, except these parameters correspond to a write operation. The Destroy operation is called when the stream identified by the stream handle is no longer needed. The Destroy method is defined as follows:

```

void Destroy (inout unsigned long   opID,
              in unsigned long        handle,
              in unsigned long        dispose

```

)
 raises (Exception);

5 The Destroy operation takes only three parameters: the *operation* identifier, the stream *handle*, and a *dispose* parameter that indicates how the stream should be disposed. In a preferred embodiment, the *dispose* parameter has three possible values indicating that the streaming completed successfully, the streaming completed unsuccessfully, or the streaming did not complete successfully because of an exception that occurred.

10 The Cancel method is used to cancel an outstanding stream operation. The method is defined as follows:

```
void Cancel ( in unsigned long    handle,
              in unsigned long    opID
              )
```

15 raises (Exception);

The Cancel method takes only two parameters: the *handle* to the stream and the *operation* identifier.

20 The Stream interface raises only one exception if an error occurs. The exception returns information regarding the streaming operation. In the preferred embodiment, exception information is returned for one or more of the following reasons: (1) the object that received the object call has no stream with the requested handle; (2) the stream has lost access to the data that was streaming due to a closed connection, file deletion or similar event; (3) a specific offset was out of bounds (larger than the file size); (4) a specific offset was out of reach for the stream implementation (Get method); (5) a "Get" method was performed on a stream object that should be written or a "Put" method was attempted on a stream object that should be retrieved; (6) a stream operation failed because of a time-out that occurred while an I/O operation occurred on behalf of the operation; and (7) Get or Put operation was performed requesting or supplying more data than the stream implementation can provide.

25

30

V. Context

As stated above, the components used by the present invention are configured as Implementation Libraries that can be loaded at different times. This configuration facilitates the creation of new components and allows the default functionality of each component to be easily changed. In order for each component to perform its function, however, each component may require specialized information to carry out its task. The present invention uses a Context structure to permit the run-time transfer of predefined information ("context") between components.

In a preferred embodiment, context is passed from component to component using the context structure shown in Figure 4. The structure is an octet string consisting of name/value pairs, where the name is an alphanumeric string and the value 197 is an any type defined in IDL. In addition, the string contains the length 191 of the context element, the length 193 of the entire string, and the length 195 of the name. The value 197 is an "any" type as defined in IDL and encoded using the Presentation Conversion Utilities ("PCU"). A PCU encoding is a presentation-independent encoding of an IDL-defined data structure. The Presentation Conversion Utilities are described below in Section IX.

Context is implemented in the present invention as an abstract data type containing a variable amount of data. A component creates context and places information placed in the context structure, such as particular information about the component. The information is provided as part of the factory. The factory is then completed with context and shipped to the navigator. The navigator's decision is based solely on the context information. The information provider may or may not use the information provided in the context. If trading is involved, the trader uses the information.

To manipulate and extend the context, a series of CEE routines are provided. These routines allow the structure of context to change without affecting the applications that utilize the context. Many of these routines take a pointer to a context buffer. The Put functions take a name and a descriptor of a data structure as written in compact IDL notation ("CIN").

A context structure is initialized by a component using the function IM_Context_Initialize, whose prototype is as follows:

IM_Context_Initialize (

in char *context,
in long size);

5

The *context* parameter is an allocated context structure as described above. The *size* parameter is the buffer space available in the context structure. This function initializes the context structure, preparing it for use by other functions.

To store an element into a context structure, the component can call the function

10 IM_Context_ElementPut, which returns a value IM_Context_RSXXX:

IM_Context_RSXXX

IM_Context_ElementPut (

in char *context,
in long contextSize,
15 in const char *name,
in const char *cin,
in void *buffer);

20 The *context* parameter contains an initialized context structure. The size of the *context* structure is specified in *contextSize*. The name of the element to be stored is specified by the *name* parameter. The *cin* parameter is a string describing the data type of the value of the element. The *buffer* contains the element's value that will be stored upon a successful return from the function. This buffer has the structure of an IDL "any" type.

25 The function also returns information regarding the success or failure of the operation. The IM_Context_RSXXX value indicates whether or not the operation succeeded. If the operation did not succeed, the return value indicates that (1) an element with the name supplied was not found in the context; (2) the context supplied did not contain a valid context structure; or (3) the context supplied did not have room to store the element.

30 Three functions, IM_Context_StringPut and IM_Context_LongPut are used to store a string or a long, respectively, into a context structure. These functions are defined as follows:

IM_Context_RSXXX

IM_Context_StringPut (

```

5      in    char      *context,
      in    long      contextSize,
      in    const char *name,
      in    char      stringValue);

```

IM_Context_RSXXX

10 IM_Context_StringPut (

```

      in    char      *context,
      in    long      contextSize,
      in    const char *name,
15      in    char      longValue);

```

In these functions, the *context* parameter is an initialized context structure. The size of the structure is specified by the *contextSize* parameter. The name of the string or long to be stored is specified by the *name* parameter and the value of the string or long to be stored upon a successful return from the function is specified by *stringValue* or *longValue*. The function returns IM_ContextXXX to indicate the success or failure of the operation. The return value, IM_Context_RSXXX, is identical to the return values for IM_Context_ElementPut.

To retrieve elements from the context structure, the application or component can call IM_Context_ElementGet, IM_Context_StringGet, or IM_ContextLongGet, which are defined as follows:

IM_Context_RSXXX

IM_Context_ElementGet (

```

30      in    const char *context,
      in    const char *name,
      out   any      *buffer,

```

```

        in    long    bufferSize,
        out   long    *bufferUsed);

```

IM_Context_RSXXX

5 IM_Context_StringGet (

```

        in    const char    *context,
        in    const char    *name,
        out   char          stringValue,
10      in    long          stringSize);

```

IM_Context_RSXXX

IM_Context_LonggGet (

```

15      in    const char    *context,
        in    const char    *name,
        out   long          *longValue);

```

20 The parameters are similar to their Get counterparts. The name parameter is the name of the string or long or element to be retrieved from the context structure.

Elements can also be deleted from the context structure through IM_Context_ElementDelete, which is defined as follows:

IM_Context_RSXXX

IM_Context_ElementDelete (

```

25      in    char          *context,
        in    const char    *name);

```

30 In this function, context is an initialized context structure and name is the name of the element to be deleted. The return value for each of these operations, IM_Context_RSXXX, is identical to the previous context functions.

VI. Abstractions

A. Adapter

Each abstraction—Adapter, Gateway, Server, or Trader—is composed of one or more actual components. The Adapter abstraction waits for incoming requests from a requestor (such as a request from the Web Browser 49 on the client computer). The Adapter components connect information in the information delivery system to the requestor. As shown in the example of Figure 6, an Adapter may include up to six components: (1) A transporter component; (2) an authenticator component; (3) a dispatcher component; (4) an adapter component (not to be confused with the Adapter abstraction); (5) a navigator component; and (6) a transformer component.

The transporter component provides a requestor with an interface to various transport protocols on different platforms. The transporter component implements TransportFactory and TransportStream interfaces that inherit from the Factory and Stream interfaces, respectively. The TransportFactory creates a TransportStream object. The TransportStream object represents the resources necessary to establish a connection for a request arriving at the transporter component. The TransportStream interface supports the Get and Put operations. The Get and Put operations may correspond to reading and writing data or sending and receiving data. The TransportStream interface adds three additional operations: (1) Connect; (2) Accept; and (3) Disconnect.

The Connect operation is called by a component to connect a TransportStream object to a remote host and is defined as follows:

```
void Connect (inout unsigned long  opID,
              in unsigned long      handle,
              in String              path
              )
    raises (Exception);
```

The *handle* is the handle to the TransportStream object. The *path* parameter is a string that describes the connection to be made ("remote.host.com:80" for a TCP transporter or "//remote_host/pipe/named_pipe" for a Lan manager, e.g.).

The Accept operation causes the stream to wait for new connections. When a connection is established, the call completes. The Accept operation is defined as follows:

```

        void Accept( inout unsigned long  opID,
                    in unsigned long      handle,
                    out String             path)
        )
5      raises (Exception);

```

The *handle* parameter is the handle for a particular TransportStream object, as returned by the TransportFactory Create call. The *path* parameter is a string that describes the connection established.

10 The Disconnect method causes the TransportStream object to disconnect from any remote host to which it is connected. The Disconnect method is defined as follows:

```

        void Disconnect (  inout unsigned long  opID,
                          in unsigned long      handle
15      )
      raises (Exception);

```

The authenticator component authenticates an information request that arrives at an Adapter abstraction. If the request contains user and password information, the
 20 authenticator can check the provided information against the operating system to ensure that the combination is valid.

The dispatcher component of the Adapter abstraction is an Implementation Library that directs the processing of a request by dispatching the request to the proper Adapter components in the proper order. In a preferred embodiment, the dispatcher is code that runs
 25 a multithreaded statemachine. The dispatcher calls several Create operations for the Factory interfaces of various Adapter components. The dispatcher calls the navigator component to determine which information provider factory object to use for a request. The dispatcher invokes the information provider factory object. The dispatcher then streams data from the information provider stream into the transporter stream until the information provider
 30 stream indicates that no more data is available.

The Adapter abstraction includes an adapter component that provides the logic for accepting a request and translating the request into a canonical form for use by other

components. This component also provides the logic for returning a reply to the requestor. The adapter component uses a Translator library to convert requests and replies between a particular protocol and context. The Translator library contains functions to parse and generate requests and replies. The adapter component makes a parse request to the

5 Translator library. The parse request takes the request as it arrived, parses it and stores relevant values in the context (as discussed above). Similarly, a reply can be created by taking the context and producing a protocol-specific reply.

The navigator component takes a canonical request (as provided by the adapter component) and decides what information provider and trader to use to fulfill the request.

10 The navigator component implements a Navigator interface having a single Navigate operation, defined as follows:

```
void Navigate (    inout unsigned long  contextbuffer,
                  out unsigned long    factory reference,
                  out String           factoryPath
15                )
                raises (Exception);
```

The navigator provides navigation through an interface that takes a subject and a context as parameters. The navigator then modifies the context based upon navigation rules that are

20 configured into the component. The rules used by the navigator are subdivided into subjects. A subject is used as a rule in a rule tree. The navigator's configuration can contain many subjects. Navigation based upon a single subject will add additional values to the context related to that particular subject.

Navigation is based upon rules. A rule must contain a condition and can optionally

25 contain an assignment and other rules. A condition consists of atomic conditions. Atomic conditions take one of the following forms:

contextelementname - 'regular expression': This atomic condition yields true when the value of the context element indicated by contextelementname is of type string and matches the regular expression given by 'regular expression'.

30 contextelementname operator "string constant": This atomic condition yields true when the value of the context element indicated by contextelementname is of type

string and compares to the string given by "string constant". Operator is one of "=", ">", "<", ">=", or "<=".

5 contextelementname operator numeric constant: This condition yields true when the value of the context element indicated by contextelementname is of a numeric type and compares to the value given by numeric constant. Operator is one of "=", ">", "<", ">=", or "<=".

10 In addition, atomic conditions may be combined to yield more complex conditions through the operators "not", "and," and "or".

15 The transformer component transforms data from one format to another. The transformer can be positioned between a dispatcher component and an information provider. The transformer will appear to be an information provider to the dispatcher component and will appear to be a dispatcher component to the information provider. Thus, transformation is performed transparently to the various components.

20 A transformer utilizes the Stream and Factory interfaces. The transformer Factory object calls an information provider Factory object to manufacture an information stream. The transformer Factory then manufactures a transformer Stream object and attaches the information stream and a transformation function. The factory returns the transformer Stream object's reference and handle. Exceptions can be returned indicating the success or failure of the operation.

25 Figure 6 shows the components of a sample HTTP Adapter. The HTTP adapter includes a TCP transporter component 155 that provides common interfaces to TCP across all platforms. The TCP Transporter factory creates a socket and optionally binds the socket to a port. The TCP Transporter Stream uses the Accept operation to listen and accept socket calls on the port. When a request arrives, the Accept call completes. The Connect operation is used to perform a connect socket call. The TCP Transporter stream can then use the Get operation to perform a receive on the socket. The Put operation is used to perform a send on the socket.

30 An authenticator component 161 authenticates a request for information that arrives at the adapter from an application. The authenticator component can be utilized to check for password and other user-specific information. This information is checked against the

operating system. The dispatcher dispatches the request to the remaining components in the Adapter. A transformer component 157 produces Hypertext Markup Language ("HTML") from different types of textual information.

5 The adapter component 153 uses the TCP transport to accept HTTP requests and write their replies back over the network. The adapter uses a Translator library containing functions to parse and generate HTTP requests and replies. The adapter makes a parse request that takes the HTTP request header and stores it into context elements.

10 A navigator component 159 decides, based upon certain configuration information, what information provider or trader should be used to fulfill the request. The functioning of each of these components will be described in detail below following the description of the interfaces implemented by each component.

B. Trader

15 The trader 150 includes only one component, termed the trader component which implements Trader Factory and Trader Registration interfaces. The dispatcher component of the Adapter calls the trader's Factory object. The trader component uses a load-balancing algorithm to select one information provider out of the many running instances of information providers. The trader returns a valid stream object reference and handle for an Information Provider. At start-up, information providers register with the trader and inform
20 the trader of their capabilities for delivering information.

When the Create operation is called on a trader Factory object by the dispatcher, the trader selects an Information Provider Factory object. The trader calls the factory object for the Information Provider and returns the stream object reference and stream handle to the dispatcher. The trader can return an exception if no information provider factory object of
25 the requested type has registered with the trader or if all available information provider factory objects are busy.

The Trader Registration interface allows information providers to register with the trader. The trader can then select an information provider based upon the information provided to the trader by the dispatcher component of the Adapter. The Trader Registration
30 interface includes a Register New Information Provider operation, an Un-register Information Provider operation, and a Change Registration Information operation. The Register New Information Provider call is used by information providers to register

themselves for trading. Each information provider supplies its name (e.g., "File Server, "Gopher Gateway", etc...), its object reference, the maximum number of streams that its factory is configured to create, and any other information that can assist the trader in performing a load-balancing algorithm. The information provider calls the Un-register
5 Information Provider to indicate that the information provider will shutdown or otherwise become unavailable for trading. The Change Registration Information Provider call is used to change the number of streams that a registered information provider can create or another property that affects trading.

If the trader component receives an exception upon calling an information provider
10 factory object, the trader has the possibility to recover from the failing factory. When more than one information provider factory object is available and an exception is returned for one of the factory objects, another factory object may be used. When only one information provider factory object is available and an exception indicates that the factory is temporary, the trader can be configured to wait a predetermined amount of time and retry invoking the
15 factory object.

C. Information Provider

1. Gateways

A Gateway Abstraction is used to connect external information to the Information
20 Matrix. The Gateway includes a transporter component and a gateway component. The transporter component performs the same function as the transporter component used in the Adapter abstraction. Thus, the transporter component provides an abstraction for the various possible transport protocols and the different interfaces to the protocols on different platforms. The gateway component registers itself with the trader at start-up. The gateway
25 component then provides the logic for taking a request in the canonical form (as created by the adapter component) and translating the request into an external request that can be sent out over a network using the transporter component. The gateway component also provides the logic for accepting a reply from a remote host.

The gateway component implements Factory and Stream interfaces. The Factory
30 interface contains only a Create operation. When called, a gateway Factory object creates a Stream object and stream context. The stream context can contain, depending upon its implementation, file handles, SQL cursors, additional object references, etc...

The Stream interface uses the Get, Put, and Destroy operations. The Get operation is used to request a certain range of bytes from a stream. The Put operation stores a certain range of bytes in a stream. The Destroy operation tells a Stream object that the object will no longer be used. The Stream object frees its resources and the object may be deleted. In a preferred embodiment, a stream is identified by a Stream object reference and a stream identifier. The stream identifiers are assigned by the gateway stream Factory when a call is made to the Factory. A stream identifier is a handle to an entry in a stream table. The stream table is used as a lookup table to determine the data that must be provided. The stream table contains a streams context that can be different for each implementation of a gateway. Thus, the stream context can contain file handles, socket numbers, object references, SQL cursors, or any other context that must be maintained between different calls to a Stream object.

Figure 7 shows a sample gopher gateway 139. The gateway 139 includes a gateway component 169 and a TCP transporter component 171. The gopher gateway 139 uses the same TCP transporter 171 as used in the HTTP adapter. The gopher gateway component 169 uses the TCP transporter component 171 to send gopher requests and receive replies back over the network.

2. Servers

A Server abstraction contains only a server component. The server component accepts a canonical request (as created by the adapter component) and resolves the request by accessing local information. The server implements a Factory and a Stream interface. The Create operation on a server Factory object attempts to open the requested file. If the file is successfully opened, the Factory creates a Stream object is created and initializes an entry in the stream table. The stream table contains the file name, file handle, file size, and number of bytes written and read after each operation.

The Get operation on a Stream object performs a read on the opened file. The Put operation writes to the opened file. The Destroy operation closes the file and deletes the entry in the stream table.

VII. System-Level Interactions Between Components

A. Non-trading Interactions

Referring now to Figure 8, a macroscopic view of the interaction between the various components using the above-described interfaces will be described. A more detailed description of each component's implementations is described later. All interactions between the components are based on object calls. The object calls function transparently between objects in the same capsule, between objects in different capsules and between objects on different machines.

Each of the Figures 8-13 shows object creation and interaction over a period of time (progressing from top to bottom). In particular, a start-up phase, set-up phase, stream phase, and clean-up phase are shown. Each component (or "Implementation Library" or "IL") is depicted as a box containing one or more additional boxes that represent interfaces. Object creation and destruction is denoted by a hollow circle. The life of the object is denoted by a vertical line extending from the object. An arrow with a hollow point represents object interaction. An arrow with a filled point represents object interaction that includes the passing of a context structure.

Adapters 131, 133, 135 and Information Providers 137, 139, 141, 143 may interact with or without the Trader 150. In most circumstances, a trader 150 is involved. If, however, the information delivery system includes only a single server providing a single purpose, the trader is not necessary. First, with reference to Figures 8, 9, and 10, interaction without the trader 150 will be described. This discussion assumes that a request from the WWW browser 49 arrives at the HTTP adapter 131 as shown in Figure 6 and that the gopher gateway 139, as shown in Figure 7, will be used as an information provider. It should be apparent, however, that a request may arrive at other adapters, such as a gopher adapter or NNTP adapter, for example. Further, a gopher gateway or FTP gateway or local file server may be used as an information provider.

Figure 8 shows the streaming of data between components during start-up, set-up, stream, and clean-up phases. As stated above, the components of each abstraction, preferably, are loaded in a separate capsule. In this Figure 8, Adapter components, including an adapter component 153, a dispatcher component, and a navigator component 159 is loaded in an Adapter capsule. An information provider component 139 is loaded in an Information Provider capsule.

During the start-up phase, the various components are initialized and loaded. In step 803, the implementation libraries are loaded and their initialization routines are called. The

initialization routines for an adapter component and information provider component cause a Factory object 1530, 1390 to be created. The navigator component 159 causes a Navigation object 1590 to be created. In step 805, the dispatcher component 151 creates a context structure and calls the Create operation on the adapter 153 Factory object with the context structure as a parameter. When an external request arrives at the adapter 131, from the WWW browser 49 on the client computer 40 or elsewhere, the Create operation will be completed, resulting in an adapter Stream object.

Following this initial start-up phase, the set-up phase begins. The IM computer 60 processes requests that arrive at the adapter 131. In step 807, a request from the WWW browser 49 loaded in the client computer 40 arrives at the adapter 131. The adapter 153 factory, in step 809, creates an adapter Stream object 1531 for the request. The Stream object 1531 is an interface to the web browser 49. Next, in step 811, the adapter 153 replies to the Create call from step 805. The reply to the Create call contains a reference to the newly created Stream object 1531 as well as a stream handle and all details of the request are contained in the context buffer. In step 813, the dispatcher component 151 calls the navigator component 159 with the request context created by the Create call from step 805. The navigator 159 uses the request context to select an information provider Factory object that can create a Stream object to satisfy the external request. In step 815, the navigator 159 replies to the dispatcher 151 with a reference to the selected information provider Factory object 1390. The dispatcher 151 then calls the information provider Factory object 1390 to obtain a Stream object for the requested information. In step 819, the information provider Factory object 1390 accesses the requested information. Next, in step 821, the information provider Factory manufactures a Stream object 1391 for the information. Finally, in step 823, the information provider Factory object 1390 returns a reference to the created information provider Stream object 1391 to the dispatcher 151.

During the stream phase, objects and components stream data to each other. Streaming can occur in both directions between information provider and requestor. The stream phase shown in Figure 8 describes the retrieval of information from an information provider to a requestor. The storage of data in an information provider is similar to the case of retrieval except the data flows in the opposite direction. In step 825, the dispatcher 151 calls the information provider Stream object 1391 using the Get operation to obtain a block of data from the stream. The information provider 139, in step 827, accesses the external

information source (or local information source, in the case of a server) to obtain a piece of data of the size requested. Next, in step 829, the information provider stream object 1391 returns the piece of data and a value (usually a boolean) that indicates whether there is more data available from the stream. The dispatcher 151 calls the adapter Stream object 1531 using the Put operation supplying the data returned by the information provider stream. In step 833, the Adapter returns the requested information to the external requestor (the WWW browser 49). In step 835, the adapter returns the Put call made in step 831.

When streaming is completed, as indicated by the information provider stream returning from a completed Get operation, the streams are destroyed. In step 837, the dispatcher 151 calls the Destroy operation of the information provider Stream object 1391. In step 839, the information provider returns the call. Next, in step 841, the information provider Stream object 1391 destroys itself after closing and freeing all resources acquired for the stream. The dispatcher 151, in step 843, calls the Destroy operation of the adapter Stream object 1531. In step 845, the adapter Stream object 1531 returns the call. Finally, in step 847, the adapter stream object 1531 destroys itself after closing and freeing all resources acquired for the stream object.

B. Interaction with Trader

Now, with reference to Figure 9, component interaction with the trader component 150 will be described. The start-up phase is similar to the start-up phase in the non-trader context, except that the information provider factory object 1390 registers itself with the trader object 1500 at start-up. In step 907, a request from the WWW browser 49 arrives at the HTTP adapter 131. In step 909, the adapter 153 creates a Stream object 1531 for the request that acts as an interface to the web browser 49. The adapter 153, in step 911, completes the factory operation by replying to the factory call from the start-up phase. The reply contains a reference to the created Stream object 1531 and also contains details about the request in a request context. The dispatcher 151, in step 913, calls the navigator 159 with the request context created in the start-up phase. The navigator 159, based upon the rules discussed above, uses the context to select a trader object 1500 capable of creating a stream to satisfy the external request. (If a single trader 150 is used, the navigator 159 selects the single trader.) In step 915, the navigator 159 replies with an object reference to the selected trader object 150. The dispatcher 151 then calls the trader object 1500 to obtain

an information provider Stream object for the requested information. The trader object 1500, in step 919, selects an information provider Factory object 1390 and calls the object. The Factory object 1390 is selected based upon a load-balancing algorithm. The information provider Factory object 1390 accesses the requested information in step 921 and manufactures a Stream object 1391 for the requested information in step 923. Next, the information provider Factory object 1390 returns the call with a reference to the created Stream object 1391. The trader object 1500, in step 927, returns the reference for the information provider stream object 1391 to the dispatcher. Streaming of data and clean-up is similar to the non-trading scenario. The trader 150 is transparent to the non-navigator components.

VIII. Abstraction-level Interaction Between Components

As stated above, each abstraction may carry out the above operations as required for a particular developer's needs. An Adapter abstraction, for example, may or may not include an authenticator. Accordingly, each abstraction may have a different internal structure. Now, with reference to Figures 10-13, preferred interaction between components within an abstraction will be discussed.

A. Adapters

Figures 10a-10b shows the interaction between Adapter components. In step 1003, the initialization routines of the authenticator 161 and navigator 159 create objects 1610, 1590 to satisfy requests. The transporter 155 and the adapter 53 create Factory objects 1550, 1530. In step 1005, the dispatcher 151 calls the Factory object 1530 created by the adapter 153. (The dispatcher may call the Factory object many times depending upon the number of requests that the dispatcher is willing to serve. For this example, we assume that only one request can be handled.) In step 1007, the adapter, upon receiving the call to the Factory object, calls the transporter Factory object 1550. The transporter Factory object 1550, in step 1009, creates a transporter Stream object 1551. The transporter Factory object 1551 returns the transporter Stream object reference and handle to the adapter 153 in step 1011. The adapter 151, in step 1013, calls the transporter Stream object's Accept operation. The Accept operation, as discussed above, causes the transporter stream object 1551 to wait for incoming requests.

The set-up phase begins at step 1015. In step 1015, a connect request from the web browser 49 arrives at the Adapter over the network. Specifically, the request arrives at the transporter component 155 causing the transporter stream object 1551, in step 1017, to return the outstanding Accept call. The adapter Factory object 1530, in step 1019, calls the transporter Stream object 1551 to read the incoming request in step 1021. In step 1023, the transporter Stream object 1551 returns the data to the adapter Factory object 1530. The adapter Factory object 1530, in step 1025, parses the request. If the request contains user information, this user information is validated by the authenticator in step 1027. In step 1029, the adapter Factory object 1530 creates an adapter Stream object 1531. The adapter Stream object 1531 has a context component that includes the transporter Stream object reference and handle. In step 1031, the adapter Factory object 1530 returns the Create call from step 1005 by returning the adapter Stream object reference and handle to the dispatcher 151. The dispatcher 151, in step 1033, calls the navigator 159 to find an information provider for the request. The navigator 159, in step 1035, returns a reference to an information provider object if a trader is not used. If a trader is used, the navigator 159 returns a trader object reference. The navigator may also return a transformer object reference. The dispatcher 151, in step 1037, calls the information provider Factory object (not shown) to manufacture a Stream object. The information provider returns a Stream object and handle in step 1039.

Figure 10b shows the stream phase and clean-up phase for an Adapter abstraction. In step 1041, the dispatcher 151 calls the Get operation on the information provider Stream object to obtain a block of data. The information provider Stream object returns the block of data in step 1043. In step 1045, the dispatcher 151 writes the retrieved data to the adapter stream object 1531 using the Put operation. In step 1047, the adapter Stream object 1531 writes the data to the transporter Stream object 1551 using the Put operation. The transporter Stream object 1551 writes the data over the network to the web browser 49 in step 1049. In steps 1049 and 1051, the transporter Stream object 1551 and adapter Stream object 1531 return their respective Put operations. The process of continuously calling the Get operation on the information provider stream object and the Put operation on the adapter and transporter stream objects continues until the information provider stream object returns an end-of-stream indication to the dispatcher 151.

In step 1053, the dispatcher 151 calls the information provider Stream object using

the Destroy operation. The information provider Stream object, in step 1055, returns the call and destroys itself. In step 1057, the dispatcher 51 calls the adapter Stream object using the Destroy operation. The adapter Stream object 1531, in turn, calls the transporter stream object's Destroy method in step 1059. The transporter Stream object 1551, in step 1061, ends the request by closing the network connection. In step 1063, the transporter Stream object 1551 completes the call from step 1059 and destroys itself in step 1065. The adapter Stream object 1531, in step 1067, completes the call from step 1057 and destroys itself in step 1069. The dispatcher recalls the adapter Factory object 1530 in step 1071 to obtain the next incoming request. In step 1073, the adapter 153 calls the transporter Factory object. The transporter Factory object creates a transporter Stream object 1552 in step 1075. In step 1077, the transporter Factory object returns the created Stream object reference and handle to the adapter 53. The adapter 153, in step 1079, calls the transporter Stream object's Accept operation. The transporter 55 is then ready to accept another request. When a new request arrives, the adapter enters the set-up phase once again.

B. Traders

As discussed above, a Trader includes only a trader component implementing a Trader Factory and Trader Registration interface. The trader component implements these interfaces as shown in Figure 11. During the set-up phase, a factory request arrives at the trader 150, and the trader selects an instance of a particular class of information providers that can fulfill the request. The trader, in step 1105, then calls the information provider Factory to manufacture an information provider Stream object. The information provider creates a Stream object 1391 in step 1107. Next, in step 1109, the information provider returns the information provider Stream object reference and stream identifier to the trader. The trader returns the information provider stream to the caller--the dispatcher--in step 1111. During the stream phase, the dispatcher continues to stream from the information provider stream without being aware of the intermediacy of the trader. When the information provider finishes streaming, in step 1115, it sends an end-of-stream indication. During clean-up, the dispatcher destroys the manufactured stream. Clean-up occurs without the use of the trader 150.

C. Information Providers

Figure 12 describes the information provider implementation of Factory and Stream interfaces using a stream table and an object table stored in memory 40. In the start-up phase, at step 1203, the initialization routines of the implementation libraries are called.

5 This routine creates an information provider Factory object 1390. If the trader is being used, the information provider will register with the trader by calling the trader object using the Register New Information Provider operation. In step 1205, the information provider receives a factory call from the dispatcher or from the trader. In the implementation of the information provider Factory, information in the call is used to select an information source
10 and verify that the source can access the information. In step 1208, the Factory implementation selects a stream table entry for a new stream. The implementation generates a new unique handle for the stream and writes the stream context into the table entry. Next, in step 1209, the Factory implementation creates or selects an addressable Stream object 1391. The Stream object reference and handle are returned to the caller in step 1211.

15 In the stream phase, streaming can take place in two directions using the Get and Put operations. In step 1213, a stream call arrives at the Stream object via a Get or Put Call. The call contains the stream handle. The stream object 1391 validates the stream handle and uses the handle to access the entry in the stream table in step 1214. The information in the stream table is used, in step 1215, to access the data required to complete the call. In step
20 1217, the Stream object 1391 returns the call.

In the clean-up phase, a Destroy call containing the stream handle arrives at the stream object 1391. In step 1220, the stream handle is validated and used to find the stream table entry. In step 1221, the information source is closed. In step 1223, the Destroy call completes, thus destroying the Stream object 1391 in step 1225 and freeing the stream table.

D. Transformers

Transformers may be used with Adapters and Gateways. Accordingly, streaming of data occurs differently if a transformer component is used. Figure 13 shows the interactions to and from a transformer component. Initially, in step 1303, a transformer Factory object
30 1570 and information provider Factory object are created. Next, in step 1305, a factory call arrives at the transformer factory object 1570 from the adapter component 153. The transformer Factory object 1570 calls an information provider Factory object 1390 in step

1307. The information provider Factory object 1390 creates an information provider Stream object 1391 in step 1309. In step 1311, the information provider Factory object 1390 returns the newly created Stream object's reference to the transformer Factory object 1570. The transformer Factory object 1570, in step 1313, creates a transformer Stream object 1571 and keeps the information provider Stream object's reference as part of the transformer Stream object's context. The transformer Factory object 1570 returns the transformer Stream object reference and handle to its caller.

In step 1317, a Get call arrives at the transformer's Stream object 1571. The transformer, in step 1319, calls the information provider stream object 1391 using a Get call to obtain data. The information provider Stream object 1391 returns data in step 1321. The transformer transforms this data using a transformation function. The transformation function examines the retrieve buffer for data to be transformed. If sufficient data to be transformed is contained in the buffer, the data is removed from the retrieve buffer, transformed and stored in the reply buffer. The Get call to the information provider is called again if the reply and retrieve buffer are not full and the information provider's stream object does not indicate "no more data". The transformer stream object's Get method returns the data from the reply buffer. The Stream object 1571 will indicate that "no more data" is available if there is not a "no more data" indication on the information provider's stream object, the retrieve buffer is empty, or the remainder of the reply buffer is returned. In step 1323, the data is returned to the dispatcher. If data were being stored, a Put call would arrive at the transformer in step 1317. The Put operation works similarly to the Get operation except information flows in the opposite direction.

In the clean-up phase, the Destroy method arrives at the transformer Stream object 1571 in step 1325. The transformer 157 performs a Destroy call on the information provider's Stream object 1391 in step 1327. The information provider Stream object, in step 1329, returns the call. The information provider destroys the information provider Stream object 1391 in step 1331. The transformer stream object 1571 returns the call in step 1333. The transformer, in step 1335, destroys the Stream object 1571.

IX. Presentation Conversion Utilities

Figure 14a shows a sample data structure 501 written in IDL. The structure, MyStruct, as written in IDL, includes three components: a char data type component, a

long data type component, and a boolean data type component. This type definition is contained in an IDL source file 101, for example, along with interface definitions.

A code generator parses the IDL source file and produces a header file containing a CIN description 1402. The CIN descriptor contains a series of ASCII characters that succinctly describes the structure without using identifiers (such as the name of the structure). In this example, the b3 characters identify the data structure as an IDL struct type containing three elements. The C indicates an IDL char type. The F character identifies an IDL long type and the B character identifies a boolean data type.

The function PCU_PREPARE converts the CIN description of a data type into a "prepared CIN" form which is more convenient to use at run-time than the CIN description. Prior to utilizing the routines PCU_PACK and PCU_UNPACK, the CIN description of each data structure contained in the header file 119, as generated by the code generator 111, must be "prepared". PCU_PREPARE is called once by both the client and the server. Since the call is a relatively expensive one, a single call to PCU_PREPARE during initialization saves valuable system resources. The call is preferably made during an initialization routine of the client and server application. PCU_PREPARE is defined in C as follows:

```
PCU_PREPARE (
    const char    *cinbuf,
    long          cinlen,
    long          prepbuflen,
    void          *prepbuflen,
    long          *prepbuflen,
    long          *cin_used);
```

In this function, *cinbuf* is a pointer to the buffer containing the CIN description of the data structure. The parameter *cinlen* is the size of the CIN. PCU_PREPARE returns a "prepared CIN" that will be stored in the address pointed to by *prepbuflen*. To specify a maximum length for *prepbuflen*, *prepbuflen* may be set to a particular value. The function also returns *prepbuflen*, which specifies the size of the prepared CIN contained in *prepbuflen*. A value of NULL may be passed as this parameter if this value is not required. The actual number of bytes that were read from *cinbuf* is returned in the parameter **cin_used*. NULL

may also be used as this parameter if the value of *cin_used* is not required.

PCU_PREPARE is used to create a prepared CIN, which is a table of *op_tag* data structures that describes the data type, offset, size, and alignment of the CIN-described data structure. PCU_PREPARE creates these *op_tag* structures by initially creating a *ctx* data structure used to pass context to and from each internal function in PCU_PREPARE. Using a *ctx* structure is preferred over passing individual parameters to the various internal functions. The *ctx* structure is defined in C as follows:

```

10      struct prepare_ctx_tag {
          op_def *op;
          op_def *op_table;
          op_def *op_end;
          const char    *cinptr;
          const char    *cinend;
15      long            offset;
          short          align;
          long           size;
          long           nr_unbounded;
          long           nr_ams;
20      op_def *prev_branch;
          op_def *main_union;
      };

```

The fields of the *ctx* structure are as follows. The *op* pointer points to the current operation in the prepared CIN. This pointer is incremented as PCU_PREPARE analyzes each CIN item (as described below). The *op_end* pointer points to the last possible operation in the prepared CIN plus one. The *cinptr* pointer points to the next byte to be read from the entire CIN string being prepared. The *cinend* pointer points to the last byte plus one of the CIN string being prepared. The *offset*, *align*, and *size* fields of this *ctx* structure are output parameters of process_cin_item (described below) that specify the offset, required alignment, and size of the processed field in the CIN-described data structure. A running count of the number of unbounded sequences and strings encountered in the processed CIN

string is contained in the *nr_unbounded*. A running count of the fields using the IDL "any" data type is contained in the *nr_anys* field. The field *prev_branch* points to a union branch operation previously processed. A list of This field is used to build a list of branch operations whose head is contained in the main union operation. The union operation is pointed to by *main_union*.

Once the *ctx* structure has been created, PCU_PREPARE calls PROCESS_CIN_ITEM for each character in the CIN string, TAKE_LONG for each signed long integer in the CIN string, and TAKE_ULONG for each unsigned long integer in the CIN string. PROCESS_CIN_ITEM processes a single item in the CIN string. The *ctx* structure is passed to PROCESS_CIN_ITEM. PROCESS_CIN_ITEM can be implemented in many ways. Preferably, the function uses a C-language "switch" statement containing a "case" for each possible character in a CIN string. In addition, a case statement may be used to recursively call itself to handle complex structures such as a sequence of struct types or a union of unions.

TAKE_LONG and TAKE_ULONG are used in conjunction with particular data types that are followed by numerals (number of array dimensions, etc...). TAKE_LONG extracts a signed long integer from the CIN buffer and returns the value to PCU_PREPARE. TAKE_ULONG extracts an unsigned long integer from the CIN buffer and returns the value to PCU_PREPARE. These values are used by PCU_PREPARE to create the table of *op_tag* data structures.

For each call, PROCESS_CIN_ITEM modifies the *ctx* data structure. First, PROCESS_CIN_ITEM increments the *op* pointer to ensure that the other fields of the structure correspond to the proper CIN item. In addition, the *size*, *align*, and *offset* fields of the *ctx* structure are changed. The alignment for each data type is determined based upon the following alignment rules. Base data types are aligned to their size. Thus, a short data type has two-byte alignment, a long has a four-byte alignment, etc... Struct types and union types have the same alignment as the contained field with the highest alignment requirement. Nevertheless, struct and union types, preferably, have an alignment requirement of at least two bytes. Finally, struct and union types are preferably padded to a multiple of their alignment requirement.

When each call to PROCESS_CIN_ITEM returns, PCU_PREPARE creates an *op_tag* data structure based upon the modified *ctx* structure. An array of these *op_tag*

structures is then stored in the prepared CIN buffer, `prebuf`, after calling `PCU_PREPARE`. The `op_tag` structure is a linear structure that can easily be manipulated by other functions. The structure, `op_tag`, is defined as follows:

```

    struct op_tag {
5      type_def          type;
      long              offset;
      long              align;
      long              size;
      long              nr_elements;
10     long              branch_label;
      char              is_default_branch;
      char              is_simple;
      char              reserve_XXX;
      op_def            *sequence_end;
15     op_def            *next_branch;
      op_def            *union_end;
      op_def            *default_branch};

```

The *type* parameter indicates the IDL data type of the data structure. The *type_def* type definition is an enumeration of all of the possible data types. The *offset* parameter is the offset of the component data structure from the start of the containing structure or union if the data structure is part of a structure or union. The alignment required by the data type (1, 2, 4, or 8 bytes) is specified by the *align* parameter. The *size* parameter indicates the size of the data structure in bytes including rounding. The *nr_elements* parameter is used for different purposes. For an array, the parameter indicates the total number of elements for all dimensions. For sequences, the parameter indicates the maximum number of occurrences. For strings, the parameter specifies the maximum size excluding zero termination. For structures, it indicates the number of primary fields in the structure. For unions, it indicates the number of fields in the union. The *branch_label* and *is_default_branch* parameters are for union branches only. The *branch_label* parameter contains the case label value that was specified in the IDL specification of the union, while the *is_default_branch* parameter is true if the entry describes the default union branch. The

is_simple parameter is a boolean value that is true if the data structure is of an IDL base data type and is false if the data structure is a compound type. The **next_branch* parameter is used for unions and union branches and points to the address of the next branch entry belonging to the union. In the case of a union entry, the parameter points to the first branch. For the last branch, the parameter contains the value NULL. The **union_end* parameter points to the address of the next entry following the conclusion of the final branch. The **sequence_end* parameter, used for sequences only, points to the address of the next entry following the sequenced type. The **default_branch* parameter points to the address of the default branch entry. This is used if none of the branches in the branch list matched the union discriminator. If there is no default, the value of *default_branch* is NULL. The *reserve_XXX* parameter allows fields to be added to the *op_tag* structure without causing errors in existing programs that erroneously assume the size of the prepared CIN.

Figure 14b shows the generated array of *op_tag* structures for the CIN string 502.

The first structure 1420 specifies the type, offset, size, alignment, and number of members for the MyStruct structure. The next three *op_tag* structures 1430, 1440, 1450 contain the type, offset, size, and alignment for each field in the MyStruct structure. This array of structures is stored in a buffer, *prepbuf*, that will be used by PCU_PACK and PCU_UNPACK to send structured data across a file or to a network.

Once the data structure has been "prepared" and the array of *op_tag* structures is stored in *prepbuf*, various messages stored in that data structure can be packed into a buffer and transported using PCU_PACK. PCU_PACK is used to copy a structured data type into an output buffer during transport to a file or across the network. PCU_PACK supports all IDL constructs including unions, unbounded sequences/strings and "any" types.

PCU_PACK stores the components of a structured data type into an output buffer based upon a specified format. PCU_PACK is defined in C as follows:

```
PCU_PACK (
    char          dst_integer_fmt,
    char          dst_real_fmt,
    char          dst_char_fmt,
    const void    *prepbuf,
    const void    *inbuf,
```



```

        long        outbuf_max_len,
        void        outbuf,
        long        outbuf_len);

```

- 5 The first three parameters specify how data is to be packed into the output buffer. These parameters may be caller-defined functions for performing the conversion as provided by the caller. The first parameter, *dst_integer_fmt*, specifies the format to be used for short, long and long data types in the output buffer. Examples of possible values for this format are *PCU_INTEGER_BIGENDIAN* which specifies an integer representation where the byte
- 10 significance decreases with increasing address or *PCU_INTEGER_LITTLEENDIAN* which specifies an integer representation where the byte significance increases with increasing address. The parameter *dst_real_fmt* specifies the format to be used for float and double data types in the output buffer. Sample values for this parameter are *PCU_REAL_IEEE* which specifies a floating point number representation using the standard IEEE format or a
- 15 vendor-specific value, such as *PCU_REAL_T16* which specifies a floating point number representation using the Tandem T16 format, for example. The third parameter, *dst_char_fmt* specifies the format to be used for char and string types in the output buffer. One possible value for this parameter is a character representation using the ISO Latin-1 format, a super-set of ASCII. Another possible value is EBCDIC, which permits
- 20 compatibility with IBM hosts.

The **prepbuff* parameter, as stated above, is a pointer to the address containing the prepared CIN description as returned by *PCU_PREPARE*. The **inbuf* parameter is a pointer to the address of the structured data to be stored into the output buffer. The **outbuf* parameter is a pointer to the address of the output buffer that receives the actual packed

25 data. The maximum number of bytes that can be accommodated by *outbuf* is contained in the *outbuf_max_len* parameter. The number of bytes actually written to *outbuf* is returned by the *outbuf_len* parameter. A value of NULL may be passed as this parameter if the number of bytes is not needed. If *PCU_SHORTOUTBUF* is returned by the function, then the *outbuf_len* parameter gets the required *outbuf* size.

- 30 Accordingly, to dynamically allocate memory for the output buffer, the client application can call *PCU_PACK* twice. On the first call, *outbuf_max_len* is set to zero. *PCU_PACK* will then return *PCU_SHORTBUF* and *outbuf_len* will contain the required

output buffer size. The correct amount of memory for the output buffer can then be allocated prior to calling PCU_PACK for a second time.

PCU_PACK initially creates a *ctx* structure. This *ctx* structure provides a similar function as the context structured used by PROCESS_CIN_ITEM. The structure allows large amounts of context to be shared between PCU_PACK and the lower-level routines that are called by PCU_PACK_ENGINE. This *ctx* structure is used by the underlying functions to PCU_PACK and is defined as follows:

```

10      struct pack_ctx_tag {
          char      dst_integer_fmt,
          char      dst_real_fmt,
          char      dst_char_fmt,
          char      *outptr,
          char      *outbuf_end
15      }

```

The requested destination format as specified in the call to PCU_PACK are passed to the *ctx* structure. These three fields are needed in case PCU_PACK must be called recursively to handle an IDL "any" type. The *outptr* pointer points to the next byte to be written into the output buffer. Even if the output buffer is full, the pointer continues to be updated. This allows the correct size to be returned to the caller in case of overflow. The caller can then adjust the size of the output buffer. The pointer *outbuf_end* points to the last byte plus one in the output buffer.

PCU_PACK calls an internal function, PCU_PACK_ENGINE. PCU_PACK_ENGINE receives pointers to lower-level functions that perform the actual packing of data into the output buffer. PCU_PACK also receives a pointer to *prebuf*, a pointer to the data to be packed (contained in *inbuf*), and a pointer to the *ctx* structure created by PCU_PACK. PCU_PACK_ENGINE goes element-by-element through the *prebuf* buffer and calls the appropriate lower-level function for the element based upon the *type* of the element (as specified by the type contained in the *op_tag* structure) and based upon the *dst_XXX_fmt* parameter to PCU_PACK. PCU_PACK_ENGINE provides the

address to the input buffer containing the structured data, the data type (via a CIN character), the *ctx* structure address, and the size of the data in the input buffer to pack into the output buffer (as specified by the *size* field of the *op_tag* structure).

5 PCU_PACK_ENGINE calls the appropriate lower-level function based upon the type of data contained in the *op_tag* data structure and the conversion specified on the call to PCU_PACK. The lower-level functions are known, lower-level functions that pack data either transparently or perform some specified conversion (BIGendian to LITTLEendian, e.g.). Each caller-supplied function takes data from the input buffer and places it into an output buffer. The number of bytes to be taken from the input buffer is specified by the size
10 parameter provided to the function from PCU_PACK_ENGINE. Once the data has been placed in the output buffer, the lower-level function modifies the *outptr* parameter of the *ctx* structure to point to the byte following the last byte written to the output buffer.

PCU_PACK_ENGINE uses the various lower-level functions to store data in the output buffer as follows. The structured data types in the input buffer are stored densely
15 (byte-aligned) in the output buffer in the same order as they were originally defined in IDL. The contents of any padding fields inserted by the code generator to achieve correct alignment are discarded. Similarly, the functions do not place default values in those fields.

Base type data structures are stored in the output buffer in the representation specified by the *dst_XXX_fmt* parameters on the call to PCU_PACK. Typically, these
20 parameters are set to the packer's native format without any conversion. Thus, the server application (the unpacker) would perform the actual conversion. The routines utilized in the present invention, however, permit the packer to perform a conversion of the data structures as well.

The representation of shorts, unsigned shorts, longs, unsigned longs, long longs, and
25 unsigned long longs are specified in the *dst_real_fmt* parameter to PCU_PACK. This parameter specifies the format for representing floating point numbers. The alignment of floats and doubles are specified by the *dst_real_fmt* parameter. This parameter corresponds to the format for representing integers. The representation of chars are specified by the *dst_char_fmt* parameter. The *dst_char_fmt* parameter specifies a format for representing
30 characters. Booleans and octets are not realigned. The "any" type is stored as an unsigned long specifying the length of the CIN description (whose alignment is based upon the *dst_integer_fmt* parameter), a CIN string describing the type (an unconverted ASCII string),

and the data itself (stored based upon these conversion rules).

Compound types such as arrays and unions are also realigned. Arrays are stored with no padding between elements. Sequences are stored as unsigned long integers indicating the number of occurrences followed by that number of occurrences. Any padding
 5 between occurrences is removed. The format of the long integers depends upon the *dst_integer_fmt* parameter as stated above. A string is stored as an unsigned long indicating the length of the string followed by that particular number of characters stored as chars. The format of the chars is determined by the *dst_char_fmt* parameter. Structures are stored field by field without padding. Unions are stored as a long followed by the active union
 10 branch.

On the receiving end, the server application must extract the unstructured data type and its appendages from the buffer that was packed using PCU_PACK. PCU_UNPACK then places this unstructured data into a data structure based upon the prepared CIN for the data structure. PCU_UNPACK is defined as follows:

```

15      PCU_UNPACK (
          char      src_integer_fmt,
          char      src_real_fmt,
          char      src_char_fmt,
          const void *prepbuff,
          const void *inbuf,
20      long      inbuf_len,
          long      outbuf_max_len,
          void      *outbuf,
          long      *outbuf_len,
          long      *inbuf_used);
25
  
```

The first three parameters correspond to the first three parameters of PCU_UNPACK. These parameters specify the format of data types as stored in the input buffer. These parameters are preferably identical to their PCU_PACK counterparts. The
 30 *prepbuff parameter is a pointer to the address containing the prepared CIN description as returned by PCU_PREPARE. The address of the input buffer is pointed to by *inbuf. The length of inbuf is specified by inbuf_len. The address of the output buffer is pointed to by

*outbuf. The maximum number of bytes that can be accommodated by outbuf is specified by outbuf_max_len. The parameter *outbuf_len obtains the number of bytes actually written to outbuf. The number of bytes read from the input buffer is specified by *inbuf_used. If the number of written bytes or the number of read bytes are not needed, 5 NULL may be passed as the value for these parameters.

PCU_UNPACK creates a *ctx* structure that is used to pass context around to the internal functions of PCU_PACK. This *ctx* structure is used by the underlying functions to PCU_UNPACK and is defined as follows:

```

10      struct pack_ctx_tag {
            char      src_integer_fmt,
            char      src_real_fmt,
            char      src_char_fmt,
            char      *inptr,
15      char      *inbuf_end

        }

```

The first three parameters are the formats passed to PCU_UNPACK. These parameters are 20 needed by the internal functions in case PCU_PACK is called recursively to handle an IDL "any" type. The *inptr* pointer points to the next byte to be read from the input buffer. The *inbuf_end* pointer points to the last byte plus one in the input buffer.

After creating the context structure, PCU_UNPACK calls PCU_UNPACK_ENGINE which provides the functionality for PCU_UNPACK. 25 PCU_UNPACK_ENGINE receives pointers to caller-supplied function for extracting data from the input buffer (the output buffer provided by PCU_PACK) and placing it in an output buffer. The prepared CIN buffer is also provided as a parameter. PCU_UNPACK_ENGINE goes element-by-element through the prepared CIN and stores the data into a data structure as specified by the *offset* and *size* fields of the *op_tag* 30 structures.

For each element in the prepared CIN buffer, PCU_UNPACK_ENGINE calls the appropriate lower-level user-specified function to perform the unpacking and converting.

PCU_PACK_ENGINE passes the data type and size of the data to be read from the input buffer along with the address of the output buffer to write the data.

PCU_UNPACK_ENGINE also passes the *ctx* data structure to each of the functions. Each caller-specified function then extracts the data from the input buffer and places the data into a data structure. The number of bytes to be written from the input buffer to the output buffer is determined by the size parameter. For compound types, PCU_UNPACK_ENGINE provides additional parameters to the caller-supplied functions.

If the data structure is an array, the number of elements in the array is provided. If the data structure is a sequence, PCU_UNPACK_ENGINE provides the maximum number of elements in the sequence along with the actual number of elements. If the data structure is a string, the maximum size and actual size of the string are provided to the caller-supplied functions. If the compound type is a struct data type, the number of members of the structure are provided.

Now, with reference to Figures 6 and 7, the method of the present invention will be described. Figure 6 is a flow chart of the client side of the method of the present invention. Prior to performing the method of the present invention, as stated above, compact descriptions of data structures are created by the code generator 111 and are included in the client and server stubs. This description can be created using the method described in Application No. XXX. The client and server stubs are compiled and linked into the client and server applications. Once the client stubs have been linked into the client application, in a first step 1501, the client application creates a prepared CIN description by calling the function PCU_PREPARE. PCU_PREPARE takes the CIN description of the data structure and, in step 1503, converts the CIN to an array of *op_tag* data structures by calling PROCESS_CIN_ITEM for each element of the CIN description. Each structure contains information regarding the type, offset, alignment, and size of the CIN-described data structure. A table of these structures are then stored in a memory buffer called *prepbuff*, in step 1505.

In step 1507, the client application calls PCU_PACK which packs the data structure by copying the data into an output buffer based upon the size as specified by the size field of each *op_tag*. PCU_PACK removes any alignment padding fields from the data structure and places the data structure, in step 1509, into an output buffer. Once the data structure has been packed into the output buffer, the data is transported in step 1511. The data

structure may be transported across a wire to a server application or transported to a file, such as a disk file. If another request involving the same data structure is made, this request is packed and the client application repeats steps 1505-1511 for the new request. The CIN description of the data structure need not be "prepared" again.

5 Figure 16 shows the server side of the method of the present invention. The server application, in step 1601, calls PCU_PREPARE to obtain a prepared description of the CIN. The "prepare" step is similar to step 1501 described above. The server then calls PCU_UNPACK in step 1603 to extract a structured data type and all of its appendages from the buffer that was packed using PCU_PACK. In step 1605, the structure is unpacked based
10 upon the parameters passed to the function (the same parameters passed to the PCU_PACK function). While extracting the data structure, the structure is realigned in step 1607 from the format specified in the input buffer of the server to the native alignment of the server. If another request arrives at the server, the server can call PCU_UNPACK to unpack the request without preparing the data structure.

15

X. Compact IDL Notation

Figure 17 is a flow chart depicting the generation of a CIN descriptor from an IDL data type, operation, and interface contained in an IDL source file. It will be understood that the steps of Figs. 17-19 are implemented by a CPU of a data processing system
20 executing computer instructions stored in memory. In step 1701, a code generator begins with the first line of an IDL source file and determines the data structure, interfaces, or operation described in the source file. If the described data structure is an interface, the code generator follows the directions shown in Figure 10. If the described data structure is an operation within an interface, the generator follows the directions in Figure 9. If the data
25 structure is a data type (or a parameter to an operation as discussed below), the generator generates a single character based upon a table of definitions. It should be noted that different characters may be used than those shown in the charts contained herein. Each chart contains only a preferred ASCII character. Chart A shows a preferred definition table that includes the character strings used to denote the various IDL base types.

IDL Base Type	Representation
Any	A
Boolean	B
Char	C
Double	D
Float	E
Long	F
Long Long	G
Octet	H
Short	I
Unsigned Short	J
Unsigned Long	K
Unsigned Long Long	L
Void	M

Chart A

As shown in Chart A, simple character strings are used to represent base types in a CIN descriptor.

- 5 If the data type is a compound type, such as an array or structure (struct type), a series of different steps are followed. In step 1711, a character is generated that indicates the start of the compound type. Chart B shows a sample table that includes the character strings used to denote the start of various IDL compound types.

IDL Compound Type	Representation
Array	a
Struct	b
Sequence	c
String	d
Union	e
Union Branch	f

Chart B

The particular representation of each compound type is handled differently according to type.

- 5 IDL defines multidimensional, fixed-size arrays for each base type. The array size is fixed at compile time. Arrays are represented in CIN as follows:

a nr_dimensions size_1 [size_2 , . . . size_n] base_type

The generation of characters for the array is shown in Steps 1713, 1715, and 1717. In this array representation, the character *a* represents the start of the array (as shown in Chart B).

- 10 The character *nr_dimensions* is a numeral indicating the number of dimensions in the array. The characters *size_1*, *size_2*, *size_n* indicate the size of the array in the first, second, and *n*th dimension of the array, respectively. For each element of the array, *base-type* is the descriptor for each element. The representation for the various base types is derived from the original base type table shown in Chart A.

- 15 IDL defines user-defined struct types. Each struct is composed of one or more fields of base or compound data types. Structs are represented in CIN as follows:

b nr_fields field_1 [field_2 . . . field_n]

- 20 The generation of characters to describe a struct is shown in steps 1719 and 1721. As shown in Chart B, the character *b* indicates the start of the struct. The numeral *nr_fields* indicates the number of fields in the struct. The fields in the struct are described by the descriptors *field_1*, *field_2*, *field_n*. Each field is a base or compound type. Base type fields of the struct are described as shown in Chart A. Compound types are described as shown herein (i.e., arrays are described with the start character *a* along with the number of

dimensions and the size of each dimension, etc...).

IDL defines sequences of data types. A sequence is a one-dimensional array with two characteristics: a maximum size (fixed at compile time) and a length (determined at run time). Sequences are represented as:

5 *c nr_occurrences base_type*

The generation of characters for a sequence is shown in steps 1723 and 1725. In this representation, *c* indicates the start of the sequence as shown in Chart B. The character *nr_occurrences* specifies how many occurrences of the data type are included in the sequence. The number of occurrences is then followed by the actual descriptor, *base_type*,
10 for each occurrence of the data type in the sequence. If the data type is a base type, the appropriate descriptor from Chart A is used. If the sequence consists of compound types, the descriptors are created as described herein. A sequence of sequences is possible.

IDL defines the string type consisting of all possible 8-bit quantities except null. A string is similar to a sequence of chars. Strings are represented in CIN as:

15 *d size*

The start of the string is indicated by the *d* character. The size of the string is represented by the *size* character generated in step 1727.

In IDL, unions are a cross between the "union" of the C programming language and the C "switch" statement. In other words, the union syntax includes a "switch" statement
20 along with a "case" label indicating the union branches. IDL unions must be discriminated; that is, the union header must specify a typed tag field that determines which union member to use for the current instance of a call. Unions and union branches are represented in CIN as follows:

e nr_fields f label_1 field_1 [f label_2 field_2 . . . label_n field_n]

25 The generation of characters to represent unions and union branches is shown in steps 1729, 1731, 1733 and 1735. In this representation, *e* indicates the start of a union and *f* indicates the start of a union branch within the union. The number of fields in the union is specified by the character *nr_fields*. The case label value for each field is indicated by the character *label_1*. If the field is a default, the label is omitted. The descriptor for each field, *field_1*,
30 then follows. The union fields may be either a base or a compound type. Accordingly, the field descriptor for a base type may be generated based upon Chart A. Compound types are generated as described herein.

As seen from the descriptors for base and compound types, the CIN description does not include the identifiers contained in the original IDL source file and a generic descriptor is generated. Thus, even the most complex data structures can easily be represented in string format. The following is a sample structure originally described in IDL:

```

5      union coordinate_def switch (boolean) {
          case FALSE;
              struct cartesian_def {
                  long x;
                  long y;
10          } cartesian;
          case TRUE;
              struct polar_def {
                  unsigned long radius;
                  unsigned long theta;
15          } polar;
      };
      typedef sequence <coordinate_def, 100> coordinate_list_def;

```

Using the method of the present invention, the above-described data structure would be represented in CIN as:

"c100+e2+f0+b2+FFf1+b2+KK".

In a preferred embodiment, positive numerals are followed by a plus sign ("+"). Negative numbers are terminated by a negative sign ("-"). While negative numbers may not occur frequently, their use may be required for certain data types, such as union case labels (i.e., the case discriminator may be a negative number). The CIN descriptor shown above is explained as follows: A data structure consisting of a sequence (c) with a maximum 100 elements (100+), each element consisting of a union (e) with two fields (2+). If the discriminator is zero (FALSE) (f0+) then one variant is a struct (b) containing two fields (2). The first field is a signed long (F). The second field is a signed long (F). If the discriminator is 1 (TRUE) then (f1+) the second variant is a struct containing (b) two fields (2+). The first field is an unsigned long (K). The second field is an unsigned long (K).

If an operation is to be described in CIN, then the method continues at step 1851.

Figure 18 is a flow chart depicting the steps followed in generating a descriptor for an operation. An operation descriptor is generated in CIN as:

```

    operation_synopsis
    operation_id
5   operation_attribute
    nr_params
    param_1, param_2 . . . param_n,
    nr_exceptions
    exception_1, exception_2 . . . exception_n
10  nr_contexts
    context_1, context_2 . . . context_n

```

In step 1851, the code generator generates a unique integer, *operation_synopsis*, that is derived from the string constituting the remainder of the operation's descriptor. The integer is derived by performing, for example, a cyclic redundancy check on the remaining characters in the CIN descriptor. Next, the code generator generates a unique string, *operation_id*, derived from the original IDL name of the operation. Next, in step 1855, the code generator generates *operation_attribute*, a character that indicates the attributes (none or "oneway") of the operation. For instance, if the operation has no oneway attribute, the character A is generated. If, however, the operation's attribute is oneway, the code generator generates the character B. The character *nr_params* is an integer that indicates how many parameters are included in the operation. If the operation has a non-void return type then the first parameter is the result. The *param_1* descriptor includes a character that indicates the direction of the parameter (in, out, inout, or function result) followed by the actual parameter data type. The code generator, for example, generates the characters A, B, C, and D for the directions of in, out, inout, and function result, respectively. For the specific parameters, the method returns to step 1707 in Figure 17. When the data type of each parameter has been described, the number of exceptions is identified by the integer *nr_exceptions*. The structure description for each exception is then described by returning to step 1719, which describes structures. The integer *nr_contexts* indicates the number of context names held by the operation. The names are then generated in strings, *context_1*, *context_2*, *context_n*.

The following are two sample operations originally described in IDL:

```
interface Math {
    long Add (in long x, in long y);
    long Subtract (in long x, in long y);
5    };
```

Using the method of the present invention, the above-described Add operation would be represented in CIN as:

126861413+3+ADDA3+DFAFAF0+0+

- 10 The CIN descriptor for the operation is described as follows. The beginning numeral (126861413) is derived from the remainder of the CIN by performing a cyclic redundancy check on the string "3+ADDA3+DFAFAF0+0+". The operation id contains three characters (3+). Those three characters are the string "ADD"—the operation id. All IDL identifiers must be unique and independent of case. Thus, operation id's are capitalized.
- 15 The operation does not include the oneway attribute (A). The operation includes three "parameters" (3+). Since the function returns a result, the first "parameter" is actually a function result (D). The function result is a signed long (F). The next parameter (actually the first parameter) is an in parameter (A). The parameter is of typed signed long (F). The third parameter is an in parameter (A) of typed signed long (F). There are no exceptions
- 20 (0+) and no contexts (0+).

Similarly, the CIN descriptor for the Subtract operation would be:

453399302-9+SUBTRACTA3+DFAF0+0+

Interfaces are similarly described using the method of the present invention. Figure 10 shows the generation of interface descriptors. Interfaces are defined as follows:

```
25    nr_operations
        operation_spec_1
        operation_spec_2
        operation_spec_n
```

- 30 The integer, *nr_operations* indicates how many operations are contained in the interface. Each operation is then described in *operation_spec_1*, *operation_spec_2*, *operation_spec_n* according to the above-described method for generating an operation descriptor. The code

generator 112, thus goes to step 951 in Figure 9 to describe each operation.

Using the method of the present invention, the above-described Math interface would be represented in CIN as:

2+126861413+3+ADDA3+DFAFAF0+0+453399302-

5 9+SUBTRACTA3+DFAF0+0+

The interface includes two operations (2+). The operation descriptors for the Add and Subtract operations follow the character indicating the number of operations.

As stated above, the CIN descriptors are contained in a header file that is linked into both the client and server applications. Thus, both the client and the server can make use of the descriptor as each sees fit. The CIN may be used in many ways. For example, a CIN description of a data type may be useful in creating generic functions to pack and unpack structured data types.

CIN descriptions may also be used to compare interfaces quickly. For example, a server application may have a header file containing two interfaces described in CIN. The server may then compare the ASCII string descriptions using known string comparison functions. If the server determines that the CIN descriptions are identical (or similar), the server may implement the operations of both interfaces using common methods in the server application. Thus, the CIN can be used to save time coding multiple methods for different (but similar) interfaces.

20

XI. Creating a Pickled IDL Format Data Structure

The Pickled IDL Format ("PIF") data structure is designed to be used in conjunction with IDL compilers and code generators loaded in the client memory 23 and server memory 17. The data structure is based upon an IDL source file stored in memory 23 or in memory 17. The source file may also be contained on a computer-readable medium, such as a disk. The data structure of the present structure contains a parse tree representing the IDL source file. The data structure can be stored in memory 23 or in memory 17 or on a computer-readable medium, such as a disk. The data structure that represents the source file is referred to as a Pickled IDL Format ("PIF"). The PIF file can be accessed at run-time by clients and servers that use the interfaces defined in the source file. The parse tree contained in the PIF file is an array using array indices rather than pointers. The use of array indices permits the resulting parse tree to be language-independent. The first element of the array is

30

unused. The second element of the array (index 1) is the root of the parse tree that acts as an entry point to the rest of the parse tree.

The data structure, *tu* 2001, is shown in Figure 20, and defined in IDL as follows:

```

struct tu_def {
5         sequence<entry_def>      entry;
          sequence<string>         source;
      }

```

The data structure 2001 contains a sequence (a variable-sized array) of parse tree nodes 2005, each of type *entry_def* (defined below) and a sequence of source file lines 2007. The sequence of source file lines 2007 is a sequence of strings containing the actual source code lines from the IDL source file.

Each parse tree node (or "entry") 2005 consists of a fixed part containing the name of the node and its properties as well as a variable portion that depends upon the node's type. The parse tree node is shown in Figure 21 and defined in IDL as follows:

```

struct entry_def {
    unsigned long entry_index;
    string        name;
    string        file_name;
20    unsigned long line_nr;
    boolean       in_main_file;
    union u_tag switch (entry_type_def) {
        case entry_argument: argument_def argument_entry;
        case entry_array: array_def array_entry;
25        case entry_attr: attr_def attr_entry;
        case entry_const: const_def const_entry;
        case entry_enum: enum_def enum_entry;
        case entry_enum_val: enum_val_def enum_val_entry;
        case entry_except: except_def except_def_entry;
30        case entry_field: field_def field_def_entry;
        case entry_interface: interface_def interface_entry;
        case entry_interface_fwd: interface_fwd_def interface_fwd_entry;

```

```

    case entry_module: module_def module_entry;
    case entry_op: op_def op_entry;
    case entry_pre_defined: pre_defined_def pre_defined_entry;
    case entry_sequence: sequence_def sequence_entry;
5     case entry_string: string_def string_entry;
    case entry_struct: struct_def struct_entry;
    case entry_typedef: typedef_def typedef_entry;
    case entry_union: union_def union_entry;
    case entry_union_branch: union_branch_def union_branch_entry;
10     } u;
};

```

The fixed part of the parse tree node includes *entry_index* 2105, an unsigned long which is the index for this particular entry in the parse tree. The unqualified name of the entry is contained in the field *name* 2107. The name of the original IDL source file is contained in the field *file_name* 2111. The field *line_nr* 2113 contains the line number in the IDL source file that caused this parse tree node to be created. The boolean *in_main_file* 2115 indicates whether or not the entry is made in the IDL source file specified on the command line or whether the entry is part of an "include" file. Following these fields, the parse tree node includes a variable portion--a union 2117 having a discriminator, *entry_type_def*. The union discriminator, *entry_type_def*, specifies the type of node and which variant within *entry_def* is active. *Entry_type_def* is an enumeration defined as follows:

```

enum entry_type_def {
25     entry_unused,
    entry_module,
    entry_interface,
    entry_interface_Fwd,
    entry_const,
30     entry_except,
    entry_attr,
    entry_op,

```


- ```

 entry_argument,
 entry_union,
 entry_union_branch,
 entry_struct,
5 entry_field,
 entry_enum,
 entry_enum_val,
 entry_string,
 entry_array,
10 entry_sequence,
 entry_typedef,
 entry_pre_defined
 };

```
- 15 *Entry\_type\_def* includes a list of the various types of parse tree entries. Each parse tree entry represents a constant integer that is used in the switch statement contained in *entry\_def*. For each entry, the union *u\_tag* will include a different type of structure. The first enumerated value *entry\_unused* corresponds to the value zero and is not used in determining the type of the union.
- 20 If the parse tree entry is a module (specified by the value *entry\_module*) the variable portion of the parse tree entry is a data structure including a sequence of module definitions. Each module definition is an unsigned long acting as an index in the parse tree array.
- If the parse tree entry is an interface, as specified by the value *entry\_interface*, the variable portion of the parse tree is a data structure including a sequence of local definitions
- 25 and a sequence of base interfaces from which this interface inherits. If the parse tree entry is a forward declaration of an interface (*entry\_interface\_fwd*), the union is an unsigned long containing the index of the full definition.
- Constants (*entry\_const*) are represented in a parse tree node as a structure containing the value of the constant. A union and switch/case statement are preferably used to
- 30 discriminate between the various base type constants (boolean constant, char constant, double constant, etc...) that may be included in the source file.
- Exceptions (*entry\_except*) are represented in a parse tree node as a structure

containing a sequence of fields. An attributes (entry\_attr) is represented as a data structure containing a boolean value that indicates whether the attribute is read-only and an unsigned long that indicates the data type.

If the parse tree entry is an operation (op\_def), the variable portion 2117 of the entry data structure 1105 is a data structure as shown in Figure 13. The data structure 2117 contains a boolean 2205 that indicates whether or not the operation has a one-way attribute, an unsigned long 2207 that indicates the return type, a sequence of arguments 2209 to the operation, a sequence of exceptions 2211 to the operation, and a sequence of strings 2213 that specify any context included in the operation. If the parse tree entry is an argument to a particular operation (entry\_argument), the variable portion of the parse tree entry is a structure containing unsigned longs that indicate the data type and direction of the argument.

If the parse tree entry is a union (entry\_union), it is represented in the parse tree entry as shown in Figure 23. The data structure 2117 contains an unsigned long specifying the discriminator 2303 and an unsigned long specifying the type 2305. The type is preferably specified using an enumerated list of base types. The structure 2117 further includes a sequence of the union's fields 2307. If the parse tree entry is a union branch (entry\_branch), the variable portion of the parse tree entry is a structure containing an unsigned long indicating the base type of the branch, a boolean indicating whether or not the branch includes a case label, and the value of the discriminator. Since the value is of a particular data type, preferably an enumerated list of the various base types is used to specify the value within the structure used to represent the union branch.

For data structures (entry\_struct), the variable portion of the parse tree entry includes a structure containing a sequence of the specified structure's fields. Enumerated values (entry\_enum) are represented by a structure containing a sequence of enumerated values. Enumerations of an enumerated type (entry\_enum\_val) are represented in the parse tree entry by a structure containing an unsigned long holding the enumeration's numerical value.

If the parse tree entry is a string (entry\_string), the variable portion of the parse tree entry is a structure containing the string's maximum size. A maximum size of zero implies an unbounded string. An array (entry\_array) is represented in the parse tree entry by a structure containing an unsigned long holding the array's base type and a sequence of longs holding the array's dimensions. A sequence (entry\_sequence) is represented by a structure

containing unsigned longs holding the sequence's base type and the sequence's maximum size.

For type definitions (`entry_ttypedef`), the parse tree entry includes a structure containing an unsigned long value indicating the type definition's base type. Predefined  
5 types (`entry_pre_defined`) are represented by a structure containing the data type. To specify the type, preferably an enumeration of the various base types are used.

Once the IDL source file has been described using the *tu* data structure, the data structure may be transported to a file or database using any known methods.

10

Having thus described a preferred embodiment of an object-oriented method and apparatus for delivering information, it should be apparent to those skilled in the art that certain advantages of the within system have been achieved. It should also be appreciated that various modifications, adaptations, and alternative embodiments thereof may be made  
15 within the scope and spirit of the present invention. For example, the use of a single trader has been illustrated, but it should be apparent that the inventive concepts described above would be equally applicable to a multiple-trader scenario. Indeed, each component may be replicated to support numerous protocols and configurations. The invention is further defined by the following claims.

## CLAIMS

### What is Claimed is:

1. An object-oriented method for delivering requested information stored in a first  
5 computer memory to a requestor stored in a second computer memory, the method  
comprising the steps of:
- loading into the first computer an information provider component for accessing the  
requested information;
  - creating an information provider factory object from the information provider  
10 component;
  - loading into the first computer a navigator component for selecting the information  
provider component;
  - creating a navigator object from the navigator component;
  - loading into the second memory an adapter component for accepting a request from  
15 the requestor for the requested information from the requestor;
  - creating an adapter factory object from the adapter component for creating an  
adapter stream object;
  - creating the adapter stream object for the request from the adapter factory object;
  - creating a information provider factory object from the information provider  
20 component;
  - selecting the information provider factory object using the navigator object;
  - accessing the requested information using the information provider factory object;
  - creating an information provider stream object from the information provider factory  
object; and
  - 25 delivering the requested information from the information provider component to the  
adapter component using the adapter stream object and information provider stream object.
2. The method for delivering information, as recited in Claim 1, further comprising  
the steps of:
- 30 loading, in the second computer memory, a dispatcher component for dispatching the  
request to the adapter and information provider components; and
  - calling the navigator object from the dispatcher component with request context

containing information regarding the request.

3. The method for delivering information, as recited in Claim 2, wherein the navigator object selects the information provider factory object using the request context.

5

4. The method for delivering information, as recited in Claim 3, wherein the request context is an octet structure that includes a context name and a context value.

10 5. The method for delivering information, as recited in Claim 1, wherein the step of streaming information from the information provider component to the adapter component further comprises the steps of:

calling the information provider stream object to obtain a discrete portion of the requested information;

15 accessing the discrete portion of requested information with the information provider stream object;

returning the discrete portion of requested information;

calling the adapter stream object to write the discrete portion of requested information to the adapter stream object;

20 returning the discrete portion of requested information to the requestor.

6. The method for delivering information, as recited in Claim 5, further comprising the step of indicating whether another discrete portion of requested information is available from the information provider stream object.

25 7. The method for delivering information, as recited in Claim 6, further comprising the steps of:

destroying the information provider stream object; and

destroying the adapter stream object.

30 8. The method for delivering information, as recited in Claim 5, wherein the step of creating an information provider stream object from the information provider factory object further comprises the steps of:

using information contained in the call to the information provider factory object to select an information source;

selecting an entry from a stream table stored in memory for a new stream;

generating a handle for the stream;

5 writing stream context information regarding the stream into the stream table entry;

creating the information provider stream object; and

returning an information stream object reference and the handle to the caller of the information provider factory object.

10 9. The method for delivering information, as recited in Claim 8, further comprising the steps of:

validating a stream handle included in the call to the information provider stream object;

accessing the stream table based upon the validated stream handle; and

15 wherein the step of accessing the step of accessing the discrete portion of requested information is performed using information in the accessed stream table entry.

10. The method for delivering information, as recited in Claim 8, further comprising the steps of:

20 closing the information source; and

destroying the information provider stream object.

11. The method for delivering information, as recited in Claim 4, further comprising the steps of:

25 initializing and loading into memory at least one transporter component for providing an interface to a particular transport protocol, and at least one authenticator component for authenticating the request for information;

creating a transporter factory object;

calling the transporter factory object;

30 creating a transporter stream object;

calling the transporter stream object in response to the request for information;

reading the request for information; and

returning data regarding the request for information to the adapter factory object.

12. The method for delivering information, as recited in Claim 10, wherein the step of returning the discrete portion of requested information to the requestor further comprises the steps of:

writing the discrete portion of requested information to the transporter object; and  
writing the data to the requestor;  
destroying the information provider stream object;  
destroying the adapter stream object; and  
destroying the transporter stream object.

13. The method for delivering information, as recited in Claim 4, further comprising the steps of:

initializing and loading into memory at least one transformer component for  
transforming the discrete portion of the requested information into a desired format;  
creating a transformer factory object;  
calling the transformer factory object;  
creating a transformer stream object;  
streaming the discrete portion of requested information from the information  
provider stream object to the transformer stream object; and  
transforming the discrete portion of the requested information into the desired  
format.

14. An object-oriented method for delivering information from a component stored in a memory of a computer to another component stored in the same computer or another computer, the method comprising the steps of:

initializing and loading into the memory at least one adapter component for accepting a request for information from a requestor, at least one information provider component for providing the requested information, at least one trader component for selecting one of the at least one information component, and at least one navigator component for selecting one of the at least one trader component;

registering the at least one information provider component with the trader

component;

creating a trader object for the at least one trader component;

creating a factory object for the at least one adapter component;

calling the adapter factory object;

5 creating an adapter stream object for the request from the adapter factory object;

creating a factory object for each of the at least one information provider components;

selecting an information provider factory object using a load-balancing algorithm in the trader object;

10 calling the selected information provider factory object;

creating an information provider stream object from the selected information provider factory object; and

streaming the requested information from the information provider component to the trader component to the adapter component using the adapter stream object and the selected information provider stream object.

15 15. The method for delivering information, as recited in claim 13, wherein the step of streaming information from the information provider component to the adapter component further comprises the steps of:

20 calling the information provider stream object to obtain a discrete portion of the requested information;

accessing the discrete portion of requested information with the information provider stream object;

returning the discrete portion of requested information;

25 calling the adapter stream object to write the discrete portion of requested information to the adapter stream object;

returning the discrete portion of requested information to the requestor.

30 16. The method for delivering information as recited in Claim 14, further comprising the step of:

indicating whether another discrete portion of requested information is available from the information provider stream object.



17. The method for delivering information, as recited in Claim 15, wherein the step of creating an information provider stream object from the information provider factory object further comprises the steps of:

using information contained in the call to the information provider factory object to

5 select an information source;

selecting an entry from a stream table stored in memory for a new stream;

generating a handle for the stream;

writing context information regarding the stream into the stream table entry;

creating the information provider stream object; and

10 returning an information stream object reference and the handle to the caller of the information provider factory object.

18. A computer software information delivery system for delivering requested information from one component to another component in a memory of a computer or  
15 across a plurality of computers, the system comprising:

at least one information provider component;

at least one adapter component for requesting information from the at least one information provider component via a request;

20 object-oriented means for discretely sending the requested information from the information provider component to the adapter component.

19. The information delivery system as recited in Claim 17, wherein the object-oriented means further comprises:

25 a factory interface implemented by the at least one information provider component and the at least one adapter component; and

a stream object interface implemented by the at least one information provider component and the at least one adapter component;

wherein a call to a factory object creates a stream object and wherein a call to a stream object sends or retrieves a discrete portion of the requested information.

30

20. The information delivery system as recited in Claim 18, wherein the at least one information provider includes a file server serving a plurality of local files.

21. The information delivery system as recited in Claim 18, wherein the at least one information provider includes means for retrieving the requested information from a remote host.

5           22. The information delivery system as recited in Claim 17, further comprising:  
            at least one navigator component for selecting one of the at least one information  
provider components based upon configuration information.

10           23. The information delivery system, as recited in Claim 17, further comprising:  
            at least one trader component for selecting one of the at least one information  
provider components based upon a load-balancing algorithm; and  
            at least one navigator component for selecting at least one of the trader components  
based upon configuration information.

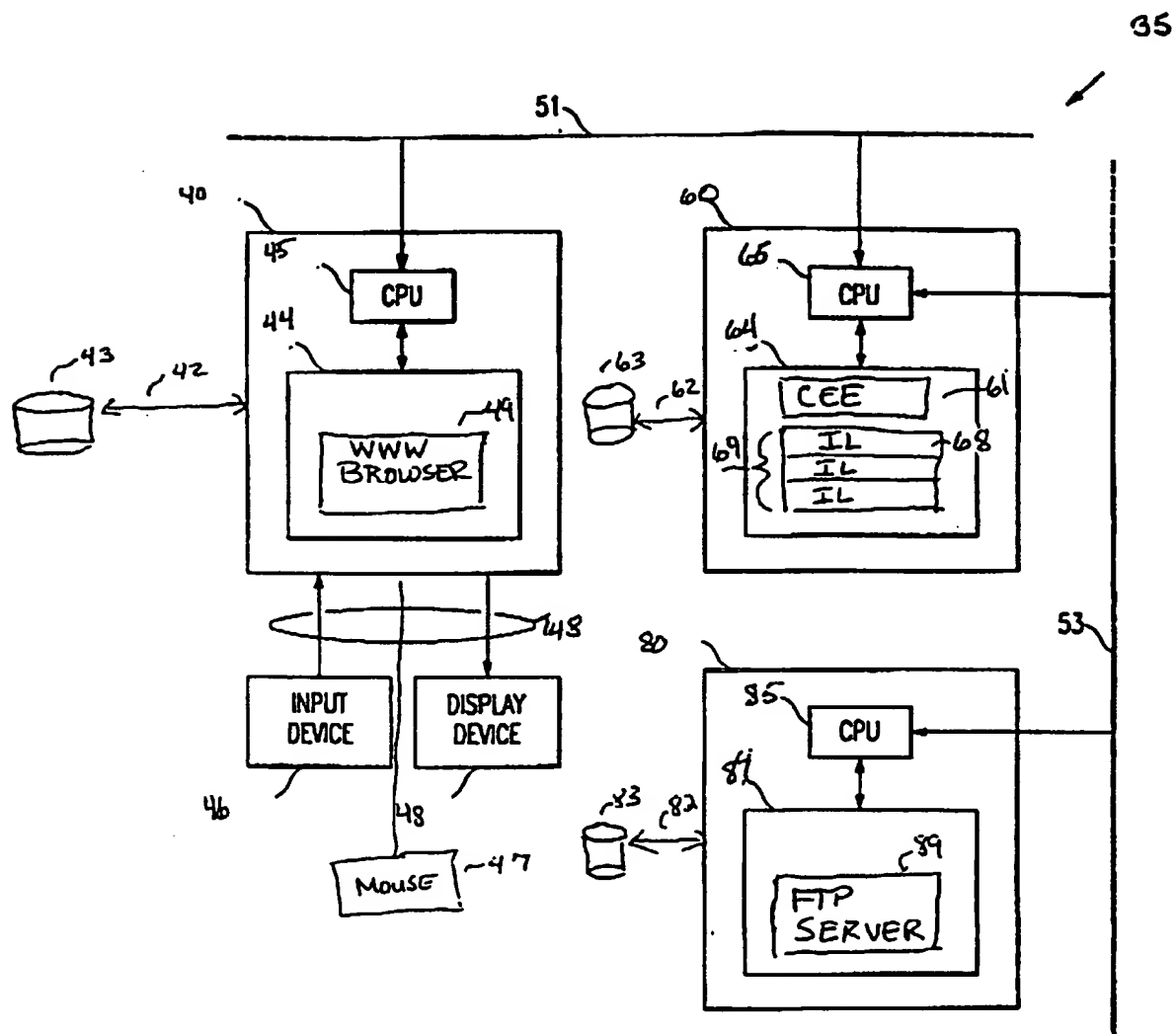


FIG. 1

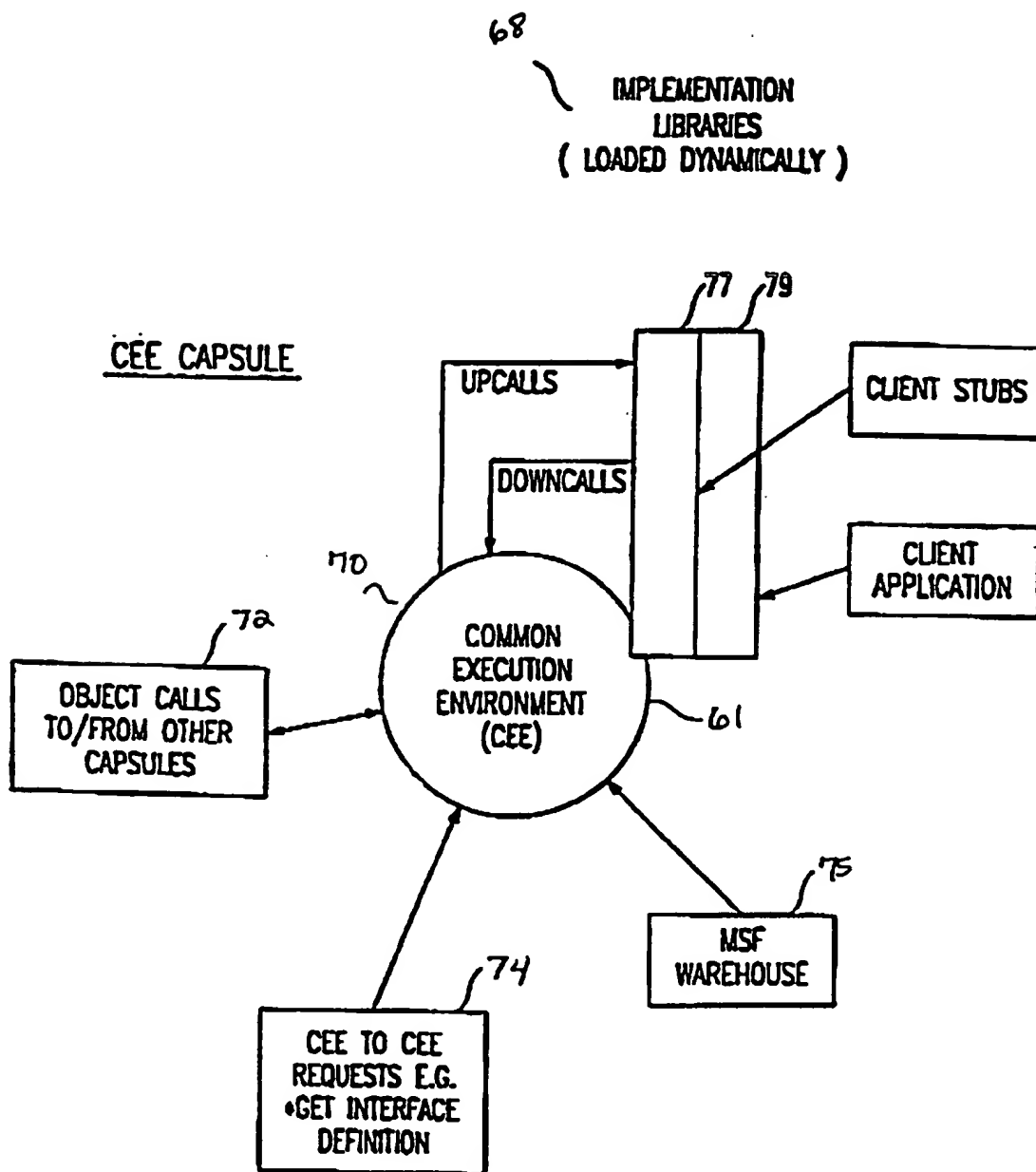


FIG. 2

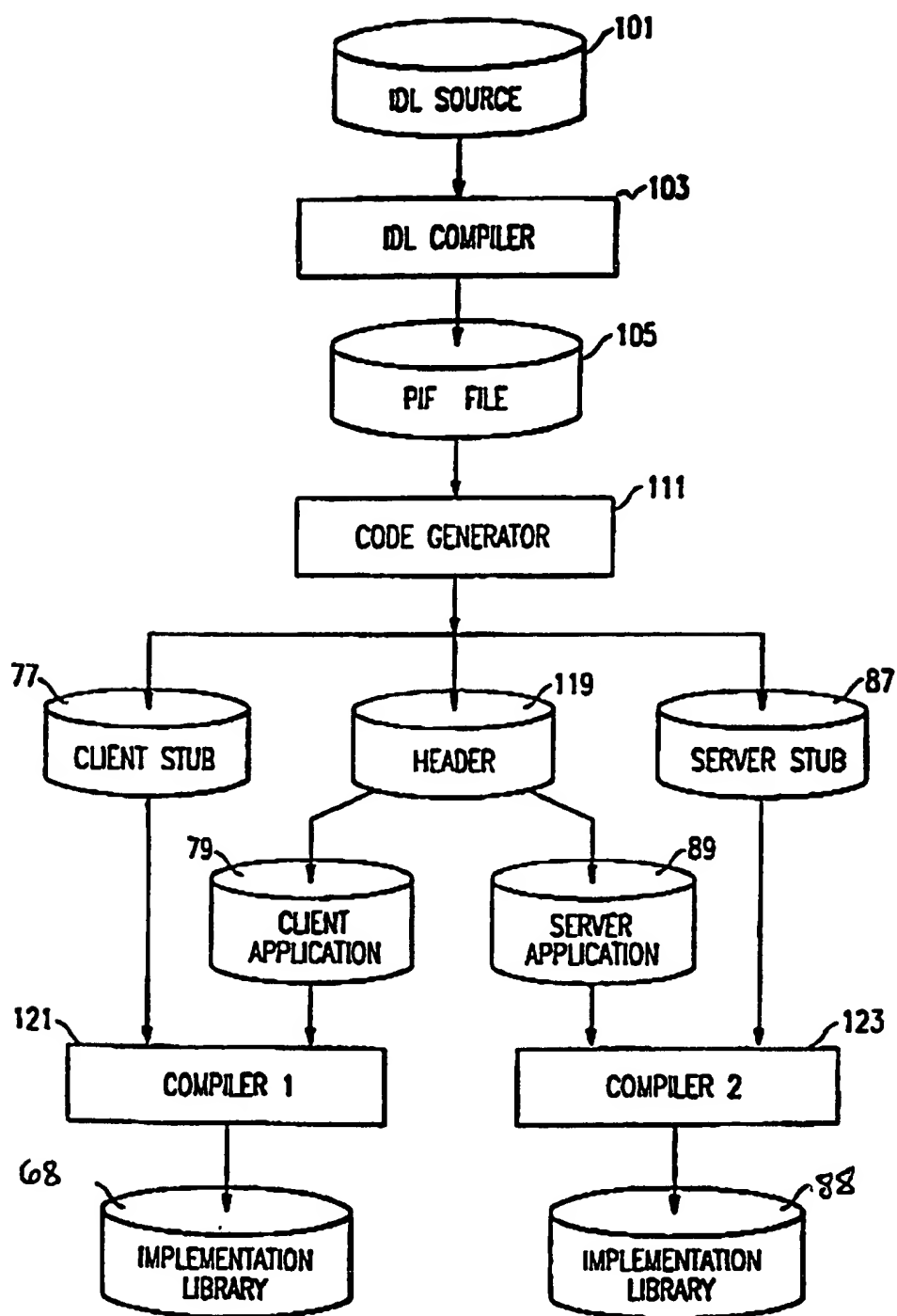


FIG. 3

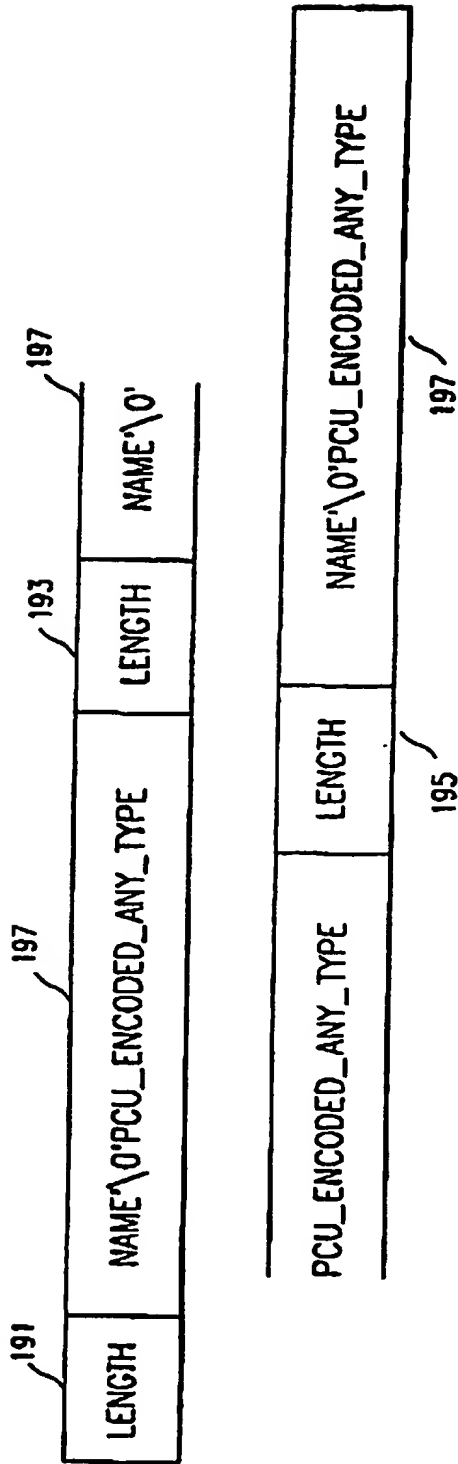


FIG. 4

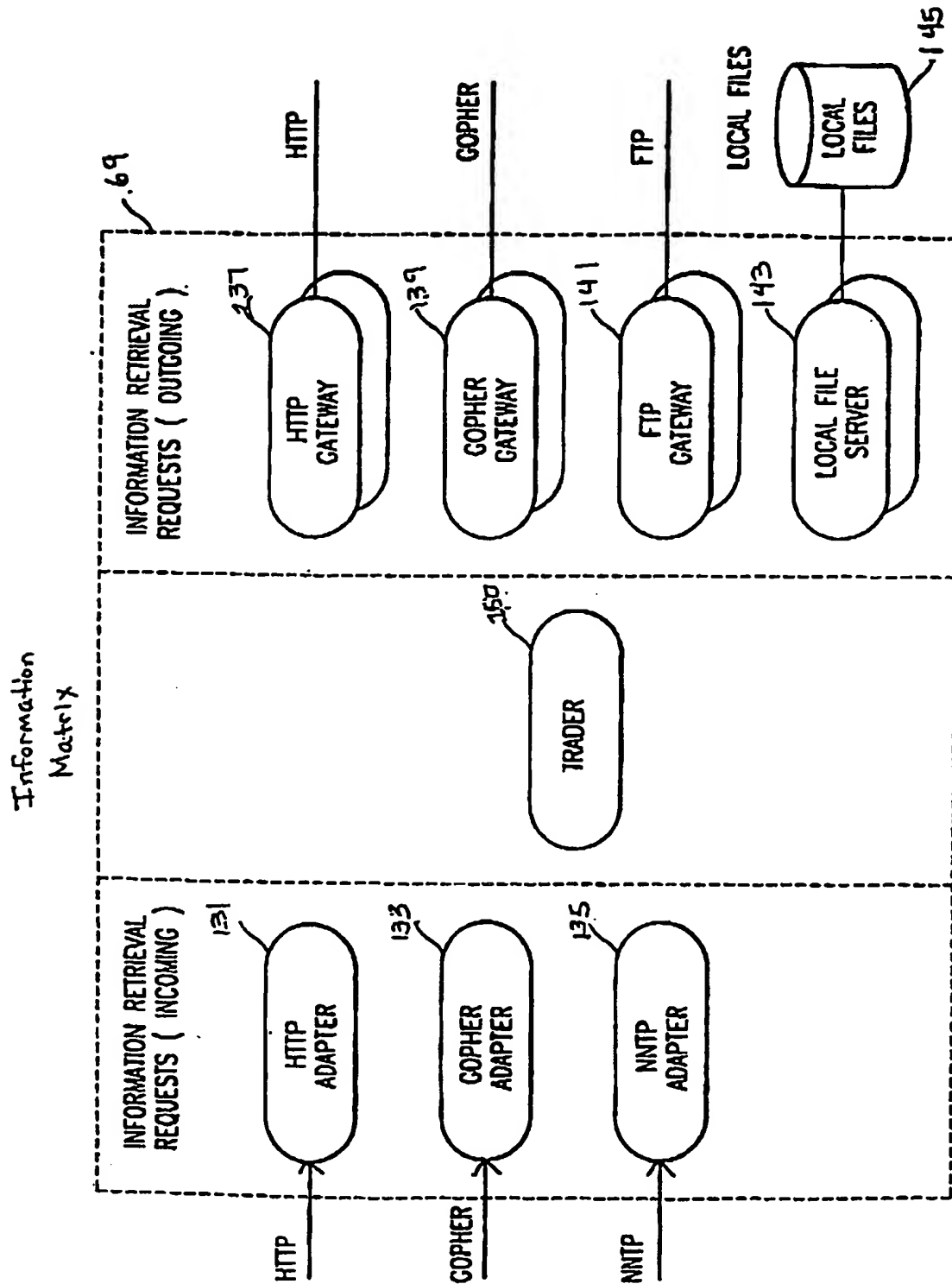


FIG 5

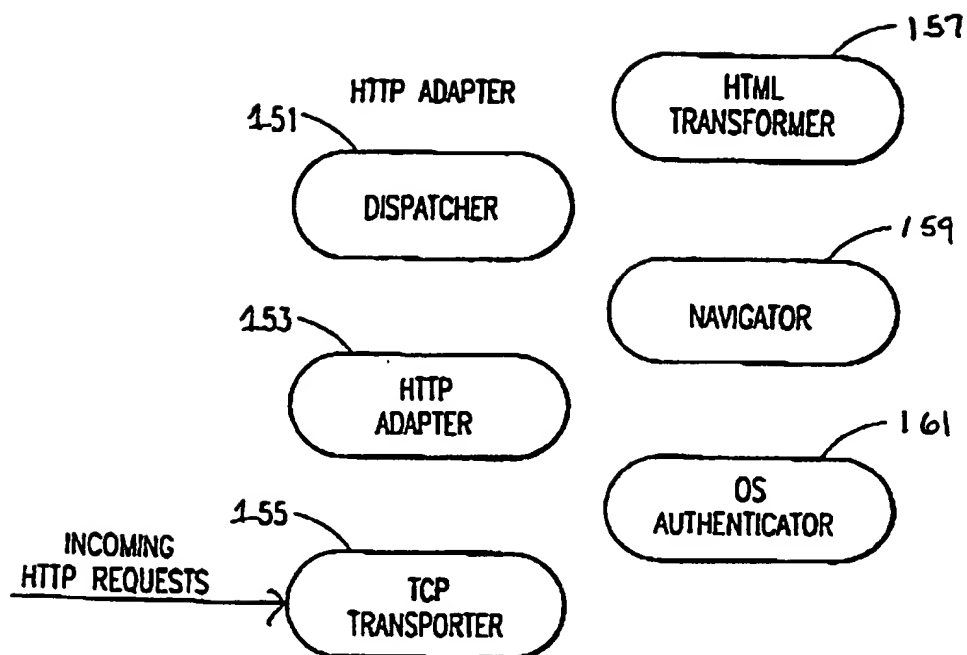


FIG. 6



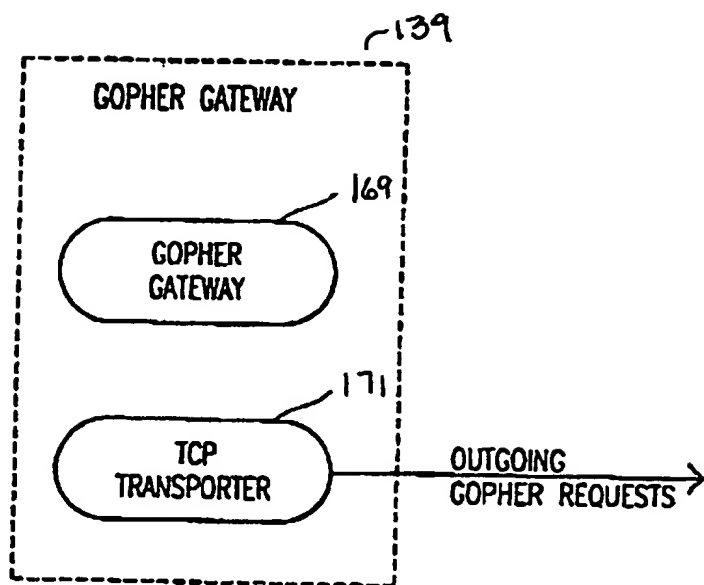


FIG. 7

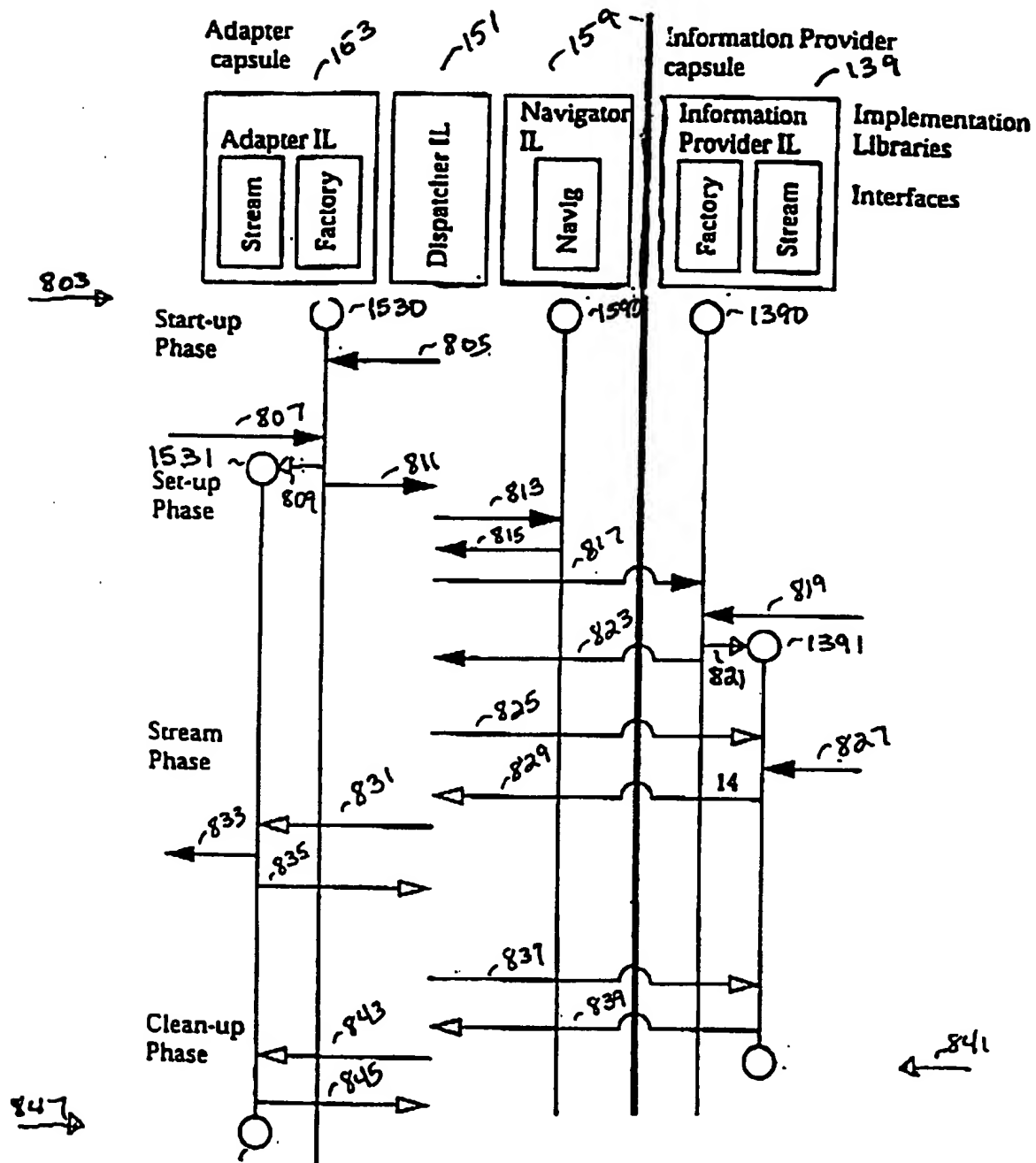


Fig. 8

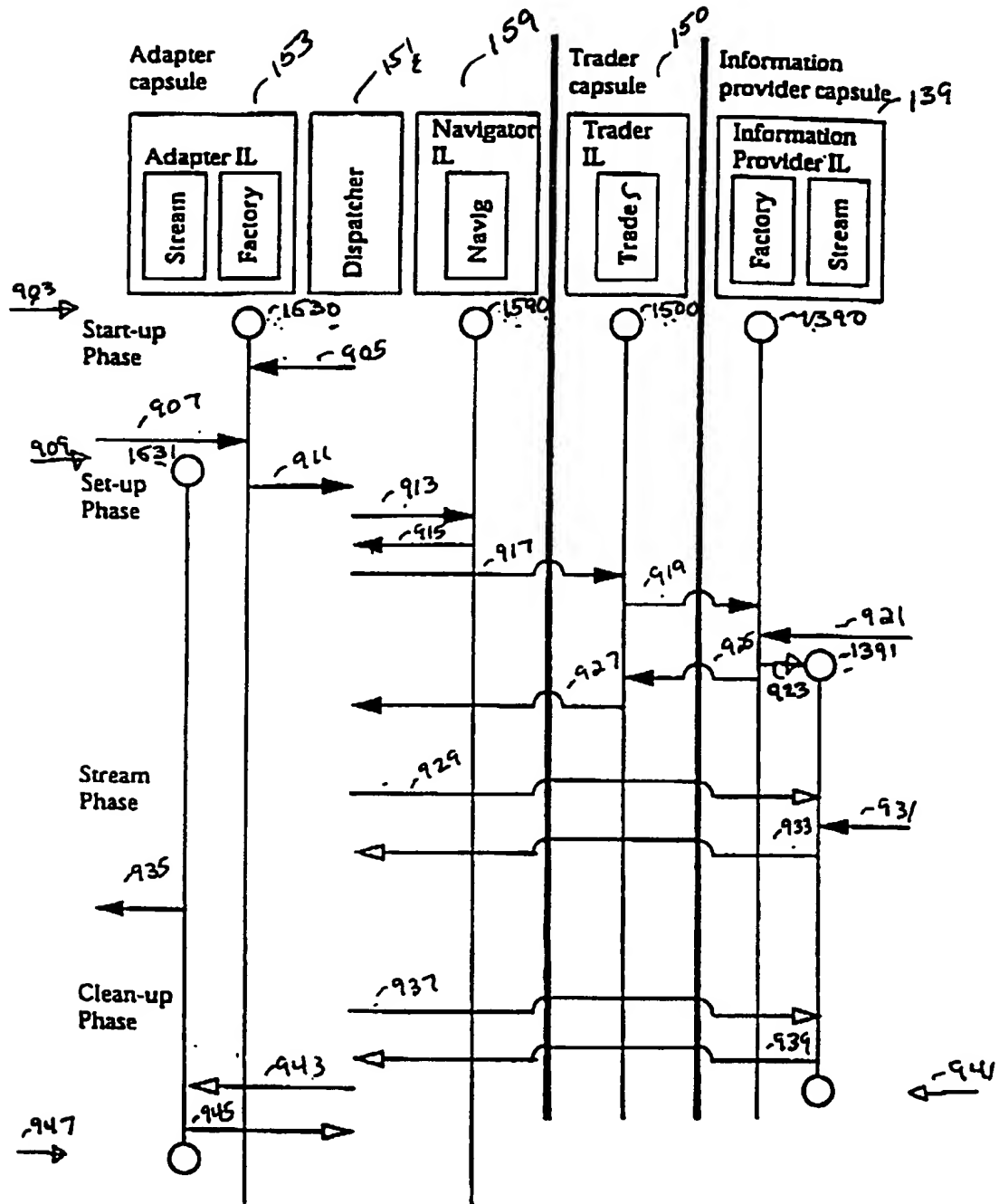


Fig. 9

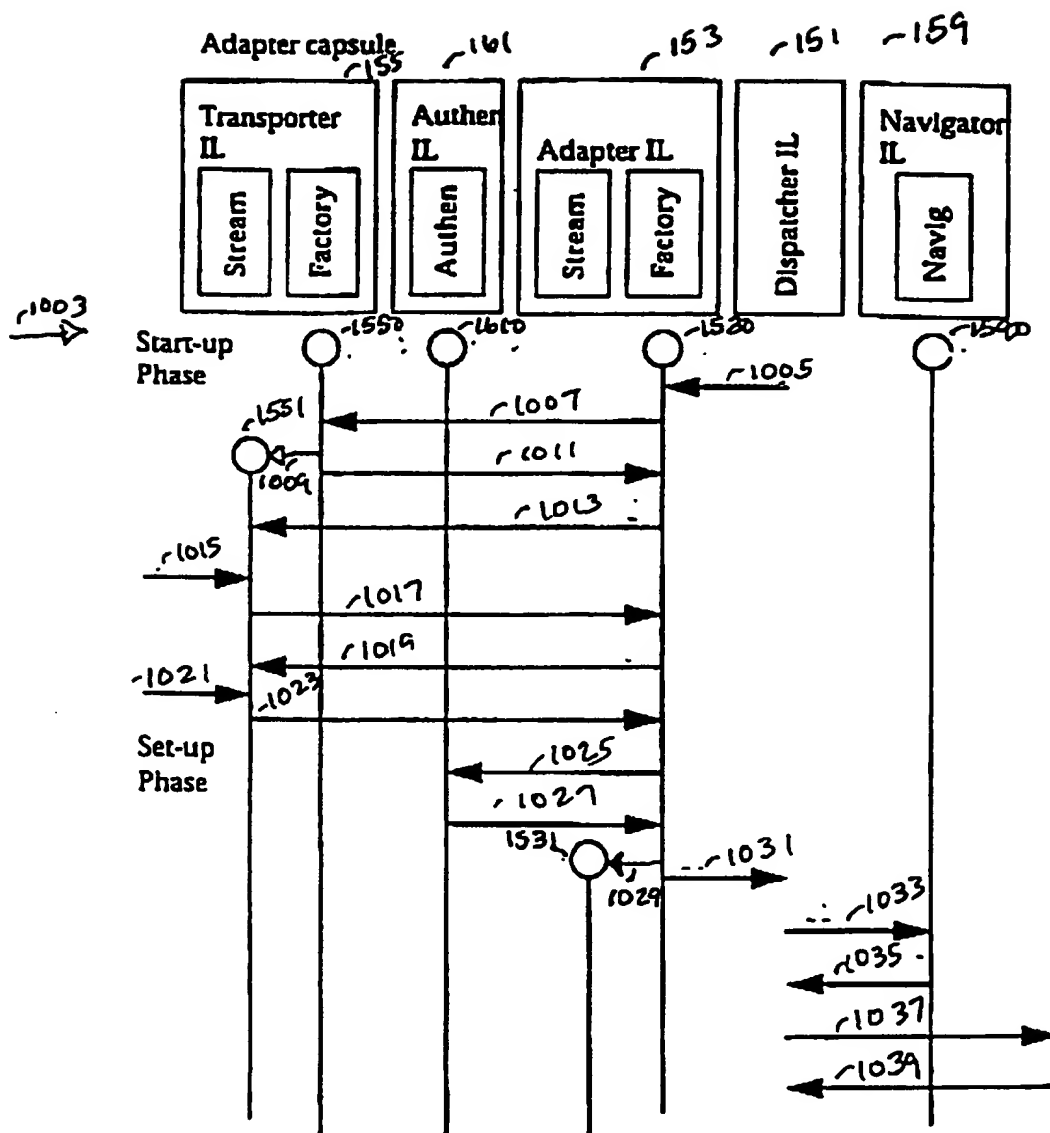


Fig. 10a

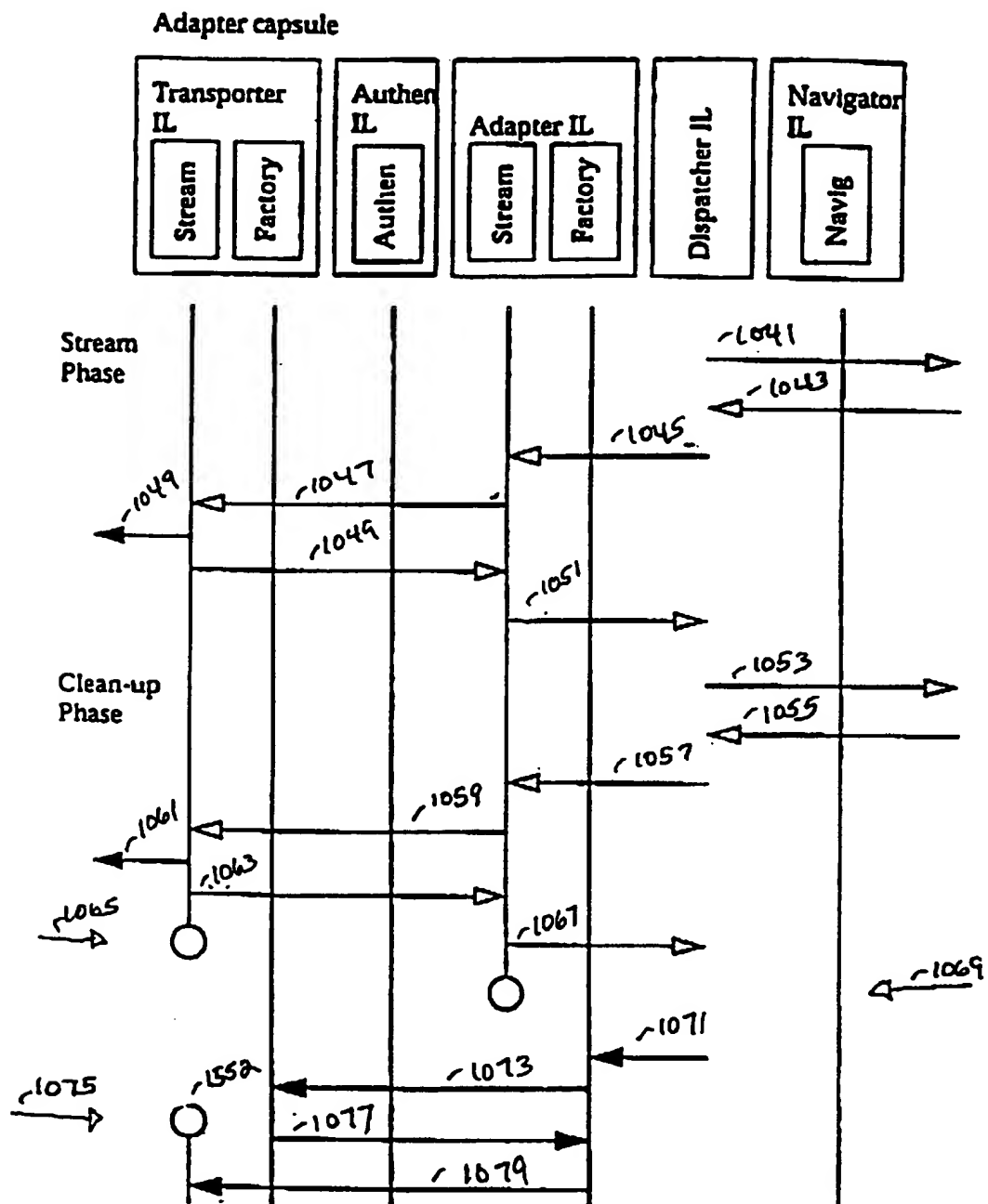


Fig. 10b

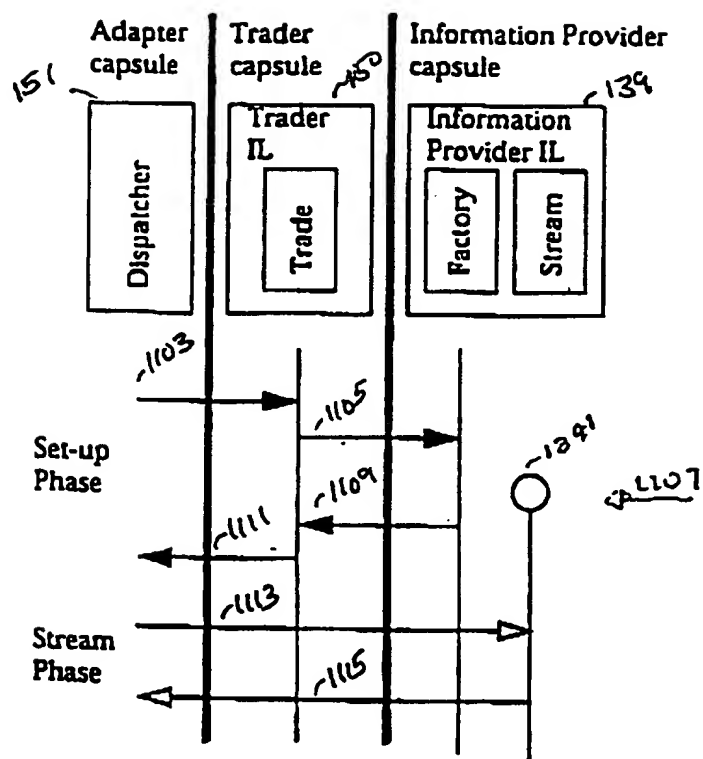


Fig. 11

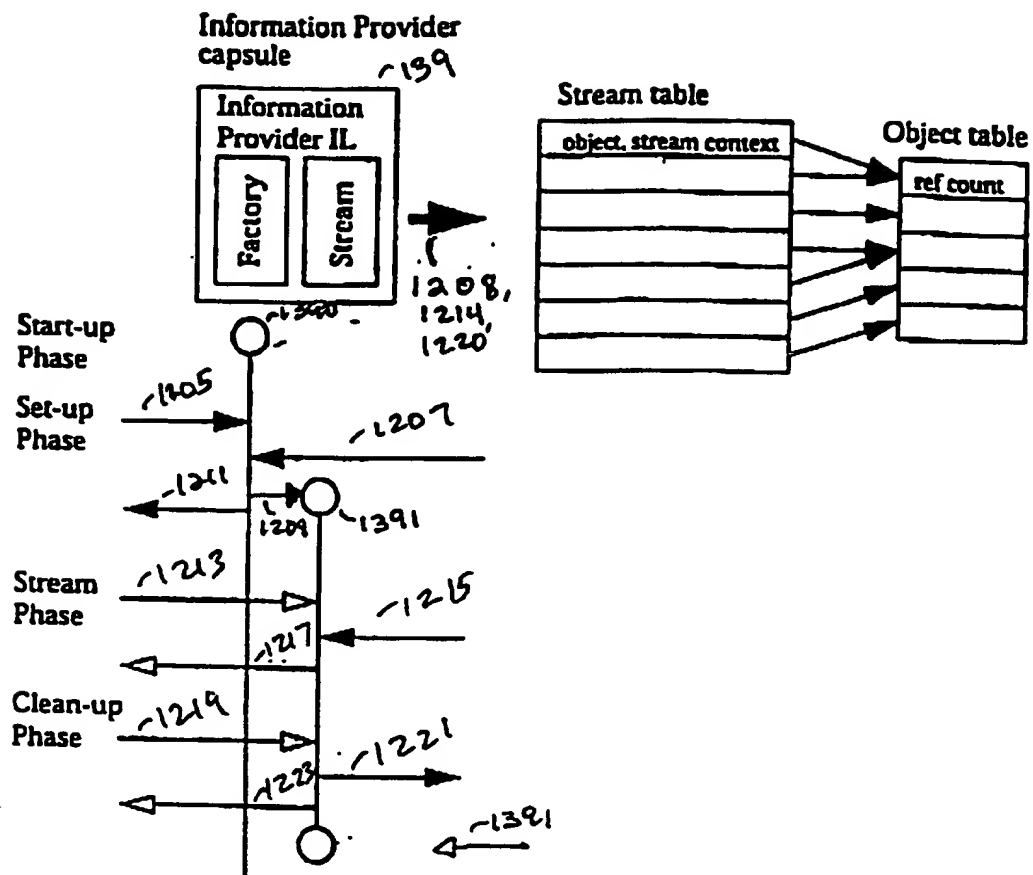


Fig. 12

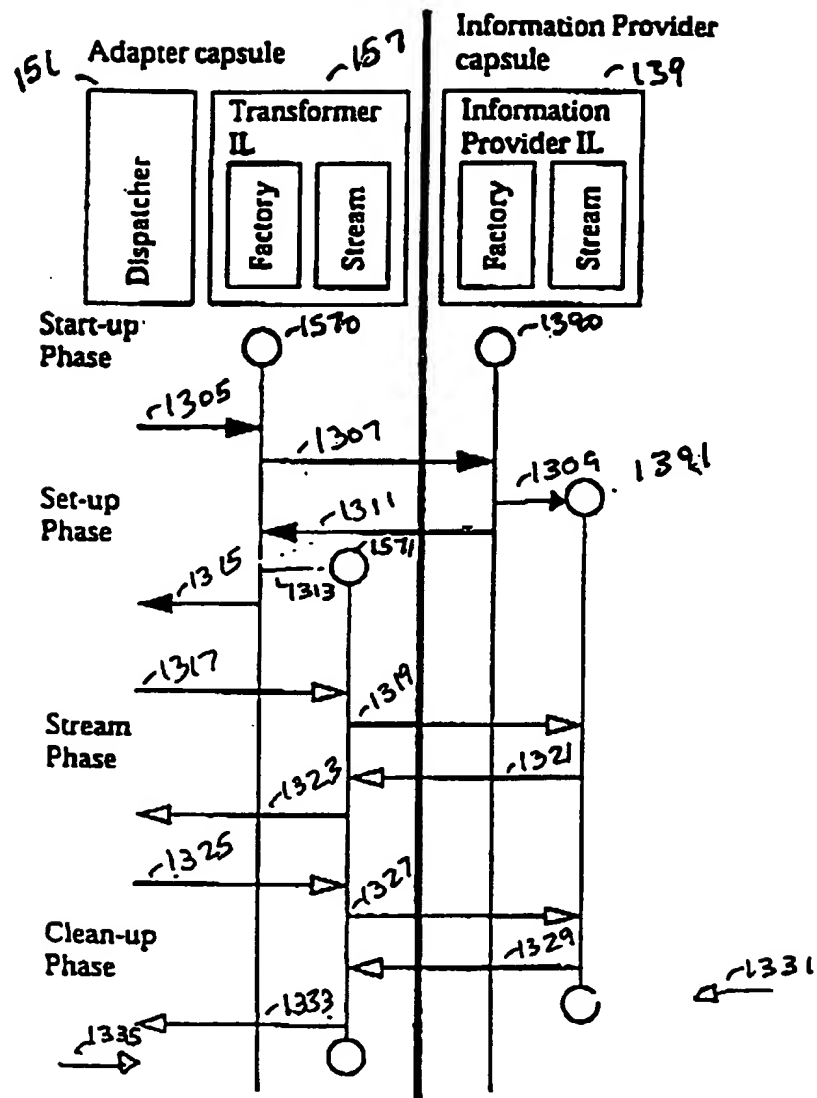


Fig. 13



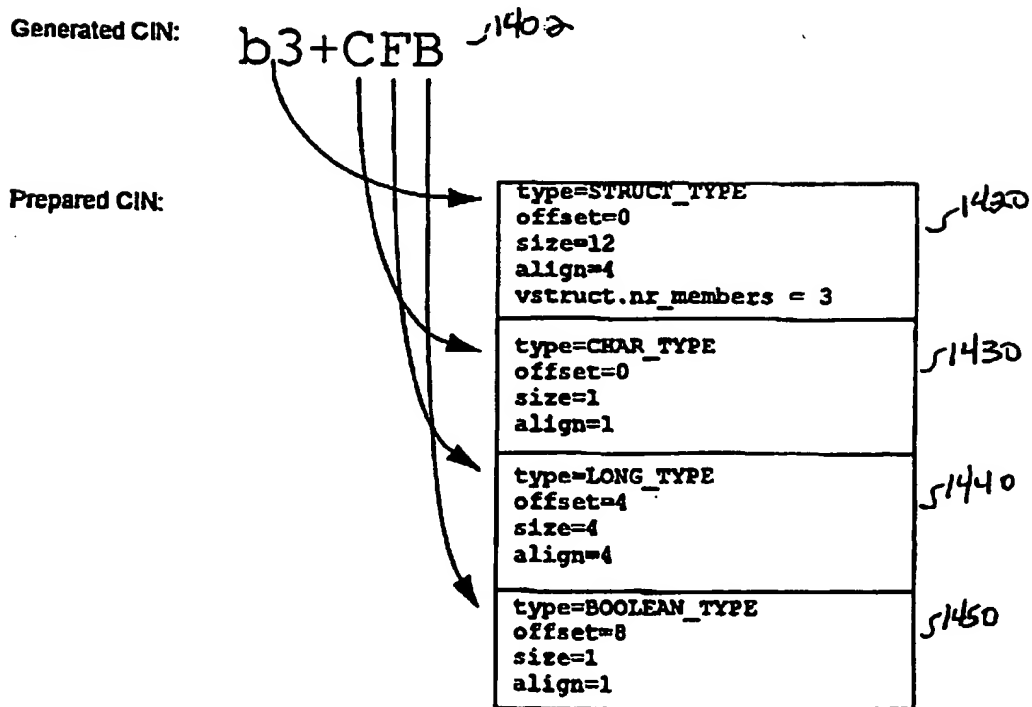
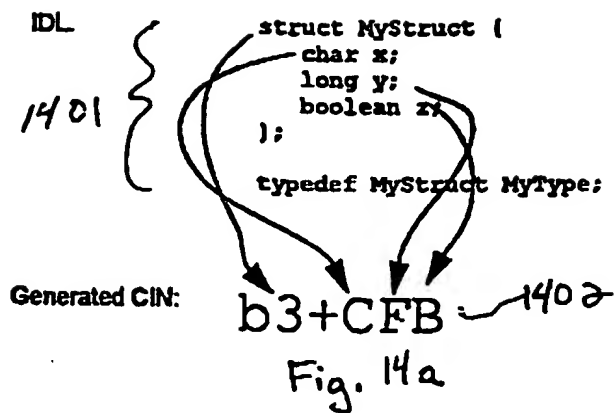


Fig. 14b

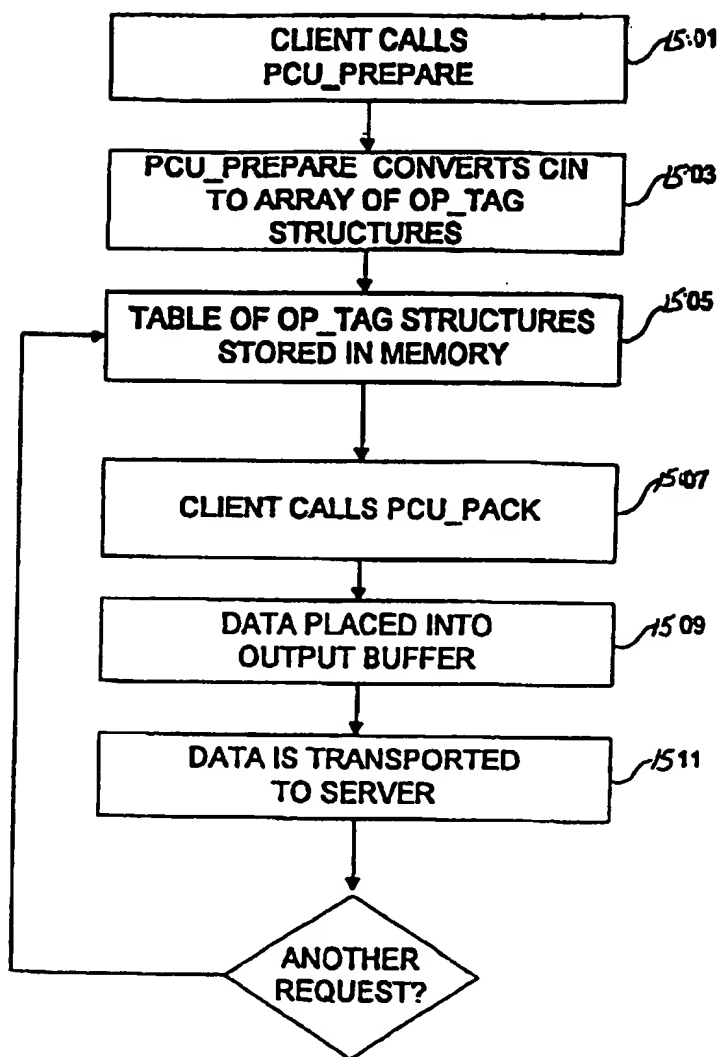


FIGURE 15

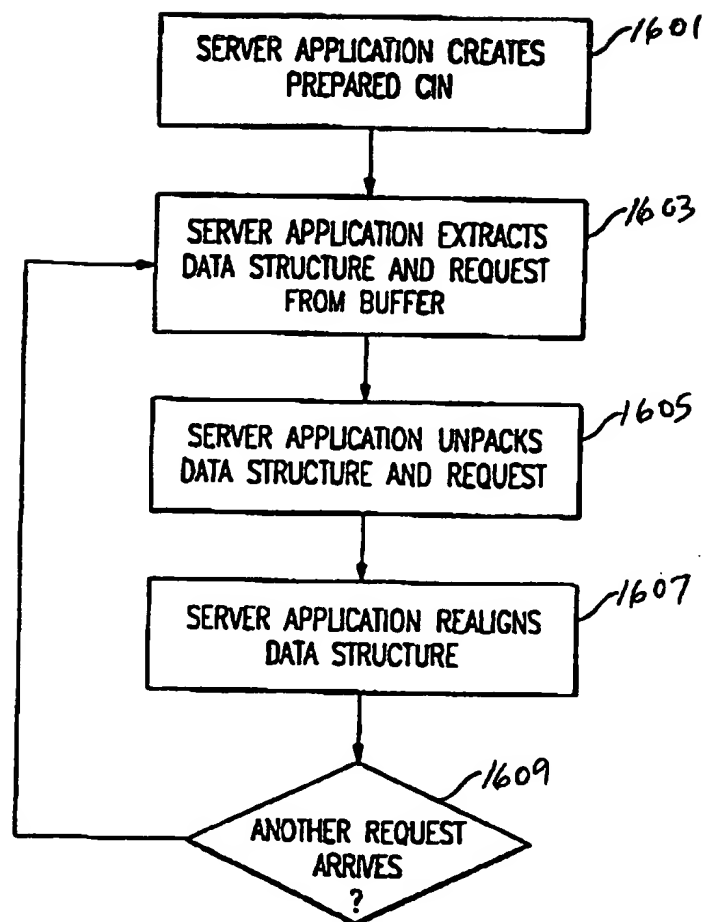


FIG. 16

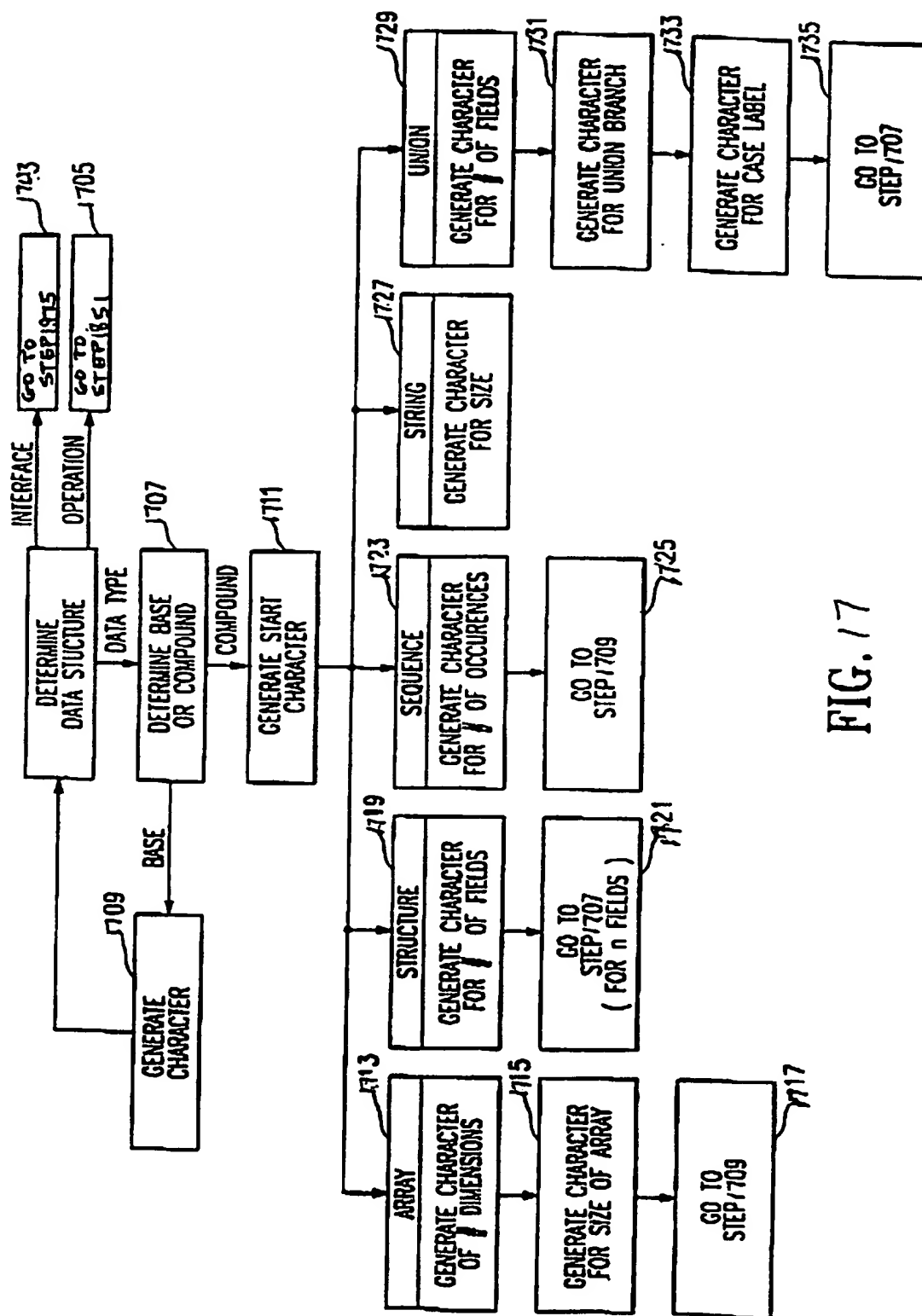


FIG. 17

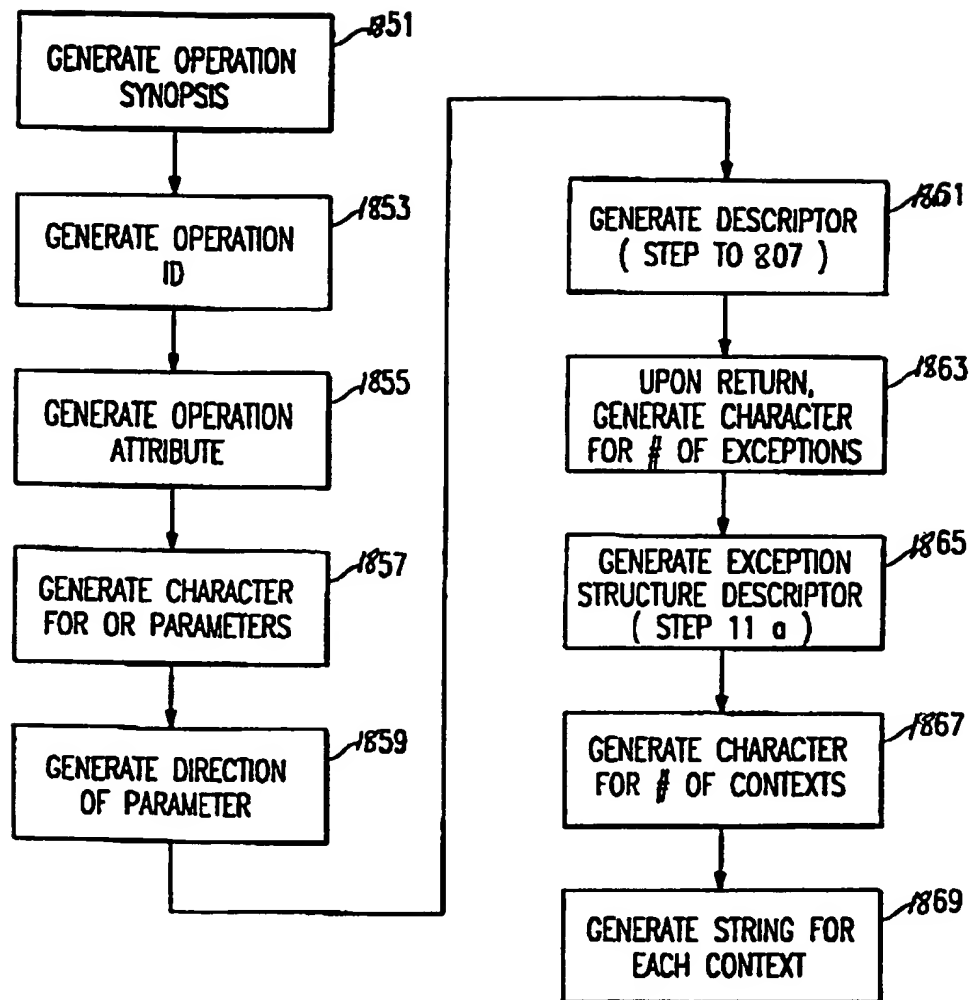


FIG. 18

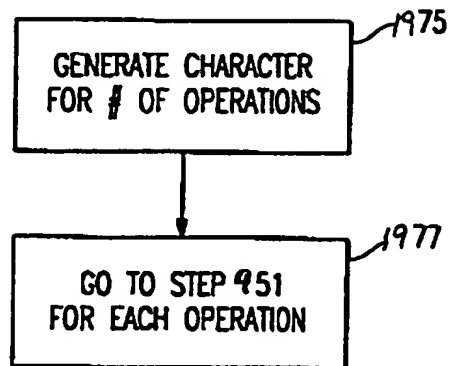


FIG. 19

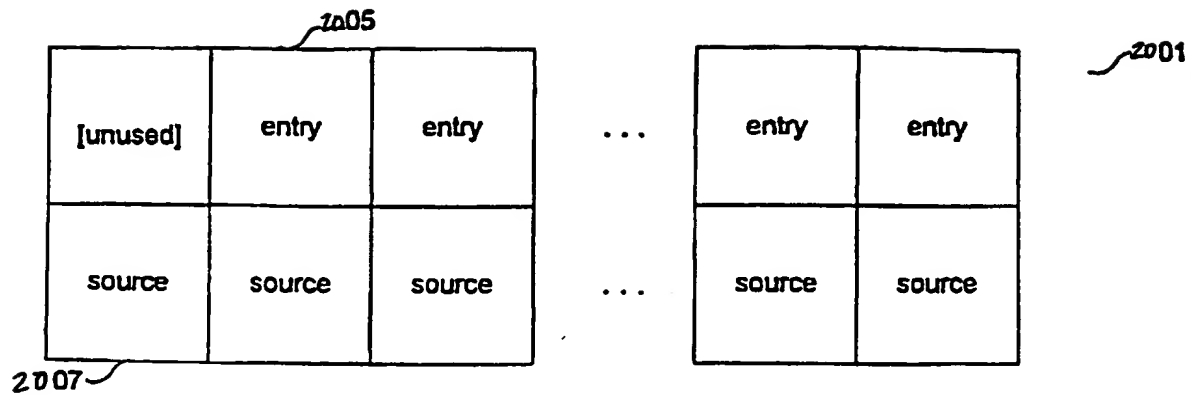


FIGURE 20

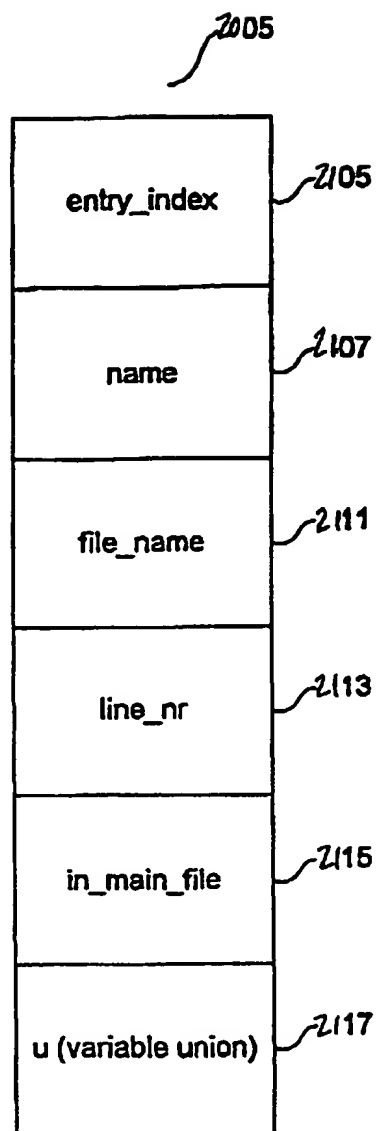


FIGURE 21



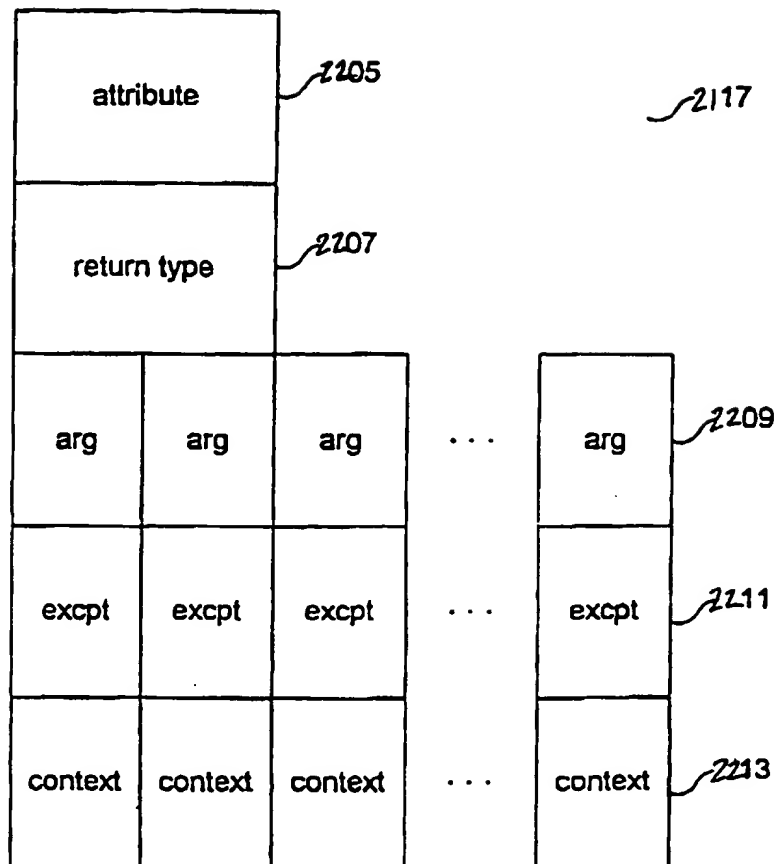


FIGURE 22

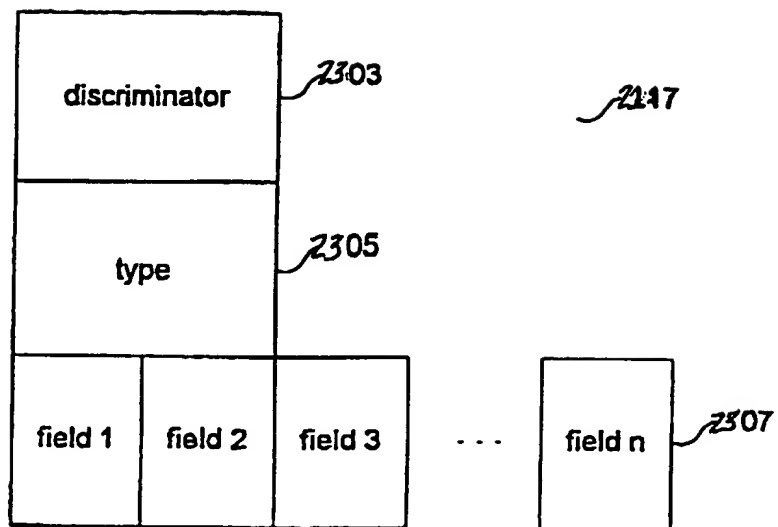


FIGURE 23

## INTERNATIONAL SEARCH REPORT

Intern. Application No

PCT/US 97/11887

A. CLASSIFICATION OF SUBJECT MATTER  
IPC 6 G06F9/46

According to International Patent Classification (IPC) or to both national classification and IP

## B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

## C. DOCUMENTS CONSIDERED TO BE RELEVANT

| Category * | Citation of document, with indication, where appropriate, of the relevant passages                                                                                                                          | Relevant to claim No. |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|
| X          | KINANE B ET AL: "Distributing broadband multimedia systems using CORBA"<br>COMPUTER COMMUNICATIONS, JAN. 1996,<br>ELSEVIER, UK,<br>vol. 19, no. 1, ISSN 0140-3664,<br>pages 13-21, XP002046137              | 18                    |
| A          | ---                                                                                                                                                                                                         | 1, 14                 |
| A          | ROBERT ORFALI ET AL.: "Client/Server Programming with OS/2 2.0"<br>1992, VAN NOSTRAND REINHOLD, NEW YORK,<br>USA XP002046139<br>Chapter 40: "SOM, OOP, and WPS Classes"<br>see page 957; figure 40.6<br>--- | 1, 14, 18             |
|            | ---<br>-/--                                                                                                                                                                                                 |                       |

☒ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

## \* Special categories of cited documents:

- "A" document defining the general state of the art which is not considered to be of particular relevance
- "E" earlier document but published on or after the international filing date
- "L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
- "O" document referring to an oral disclosure, use, exhibition or other means
- "P" document published prior to the international filing date but later than the priority date claimed

- "T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
- "X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
- "Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.
- "A" document member of the same patent family

Date of the actual completion of the international search

7 November 1997

Date of mailing of the international search report

19. 11. 97

Name and mailing address of the ISA

European Patent Office, P.B. 5818 Patentlaan 2  
NL - 2280 HV Rijswijk  
Tel. (+31-70) 340-2040, Tx. 31 651 epo nl,  
Fax: (+31-70) 340-3018

Authorized officer

Wiltink, J

# INTERNATIONAL SEARCH REPORT

International Application No

PCT/US 97/11887

## C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

| Category * | Citation of document, with indication, where appropriate, of the relevant passages                                                                                                                                                                                                                                                                                                                                                                                        | Relevant to claim No. |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|
| A          | <p>TREHAN R ET AL: "Toolkit for shared hypermedia on a distributed object oriented architecture"</p> <p>PROCEEDINGS ACM MULTIMEDIA 93, PROCEEDINGS OF FIRST ACM INTERNATIONAL CONFERENCE ON MULTIMEDIA, ANAHEIM, CA, USA, 2-6 AUG. 1993, ISBN 0-89791-596-8, 1993, NEW YORK, NY, USA, ACM, USA,</p> <p>pages 175-182, XP002046138</p> <p>see abstract</p> <p>see page 177, right-hand column, line 2 -</p> <p>page 179, left-hand column, line 25;</p> <p>figures 2,3</p> | 1,14,18               |
| A          | <p>---</p> <p>YANG Z ET AL: "CORBA: A PLATFORM FOR DISTRIBUTED OBJECT COMPUTING"</p> <p>OPERATING SYSTEMS REVIEW (SIGOPS),</p> <p>vol. 30, no. 2, 1 April 1996,</p> <p>pages 4-31, XP000585086</p> <p>-----</p>                                                                                                                                                                                                                                                           | 1,14,18               |